Automatic Software Repair: A Survey

Luca Gazzola, Daniela Micucci, *Member, IEEE*, and Leonardo Mariani, *Senior Member, IEEE*

Abstract—Despite their growing complexity and increasing size, modern software applications must satisfy strict release requirements that impose short bug fixing and maintenance cycles, putting significant pressure on developers who are responsible for timely producing high-quality software. To reduce developers workload, repairing and healing techniques have been extensively investigated as solutions for efficiently repairing and maintaining software in the last few years. In particular, *repairing solutions* have been able to automatically produce useful fixes for several classes of bugs that might be present in software programs. A range of algorithms, techniques, and heuristics have been integrated, experimented, and studied, producing a heterogeneous and articulated research framework where automatic repair techniques are proliferating. This paper organizes the knowledge in the area by surveying a body of 108 papers about automatic software repair techniques, illustrating the algorithms and the approaches, comparing them on representative examples, and discussing the open challenges and the empirical evidence reported so far.

Index Terms—Automatic program repair, generate and validate, search-based, semantics-driven repair, correct by construction, program synthesis, self-repairing

1 Introduction

Debugging software failures is still a painful, time consuming, and expensive process. For instance, recent studies showed that debugging activities often account for about 50 percent of the overall development cost of software products [1], [2].

There are many factors contributing to the cost of debugging, but the most impacting one is the extensive manual effort that is still required to identify and remove faults. In particular, the debugging process requires analyzing and understanding failed executions, identifying the causes of the failures, implementing fixes, and validating that the fixed program works correctly, that is, the problem has been fixed without introducing any side effect [3], [4], [5]. Most of these activities are executed manually or with partial tool support.

So far, the automation of debugging activities essentially concerned with the identification of the statements that are likely to be faulty [6], [7], [8], with the isolation of the specific inputs or application states that may cause failures [9], [10], [11], and with the detection of the anomalous events that may partially explain the reason of a failure[12], [13], [14], [15].

Techniques for the isolation of the likely faulty statements report to testers a list of statements ranked by suspiciousness, determined by considering the number of failing and passing test cases that execute each statement [6], [7], [8]. The intuition is that the most suspicious statements should be executed by several failing tests and few passing test cases.

• *The authors are with the Department of Informatics, Systems and Communication (DISCo), University of Milano Bicocca, Milano 20126, Italy. E-mail: {luca.gazzola, micucci, mariani}@disco.unimib.it.*

Manuscript received 6 Aug. 2016; revised 3 Sept. 2017; accepted 10 Sept. 2017. Date of publication 29 Oct. 2017; date of current version 16 Jan. 2019. (Corresponding author: Leonardo Mariani.)
Recommended for acceptance by A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2755013

Techniques that identify specific inputs and specific states that may trigger failures are often based on the well-known Delta Debugging technique [9]. The intuition is that by iteratively refining and reducing the input [10] and the state space [11], it should be possible to identify the smallest input and the smallest portion of the application state that cannot be eliminated to observe the failure. This information is relevant to understand the conditions that may trigger failures.

Finally, anomaly detection techniques can detect the operations that are executed by an application during failures but not during correct executions. These operations may explain why and how an application failed. Anomaly detection techniques usually exploit specification mining approaches [16], [17], [18], [19] to automatically generate models that represent the legal behavior of an application, and then use these models to analyze the failed executions and determine the anomalous events [12], [13], [14], [15].

All these techniques provide useful insights about the possible locations of the faults, the inputs and states responsible for the failures, as well as the anomalous operations executed during failures. However, developers must still put a relevant effort on the analysis of the failed executions to exactly identify the faults that must be fixed. In addition, these techniques do not help the developers with the synthesis of an appropriate fix.

Recently, researchers focused on a new class of approaches, namely *program repair techniques* [20], [21], [22], [23], [24], [25], [26], [27], [28], [29]. The key idea of these techniques is to try to automatically repair software systems by producing an actual fix that can be validated by the testers before it is finally accepted, or that can be adapted to properly fit the system. The benefit of using these techniques is that the fix both explains the reason of the failure and provides a possible solution to the problem, thus alleviating the effort necessary to identify and correct faults [30], [31].

Since program repair techniques have the potential to dramatically reduce debugging effort, they attracted the

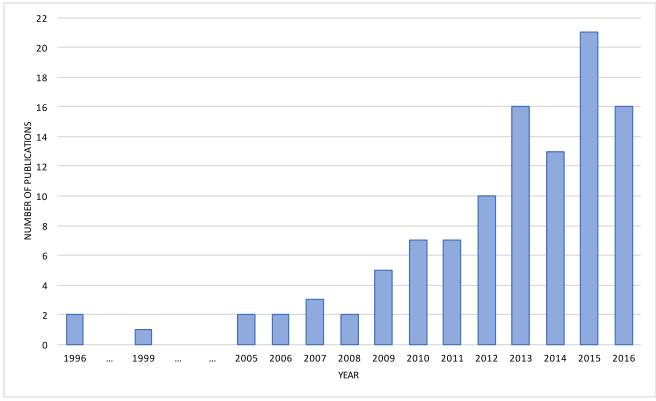

Fig. 1. Publications per year from 1996 to 2016.

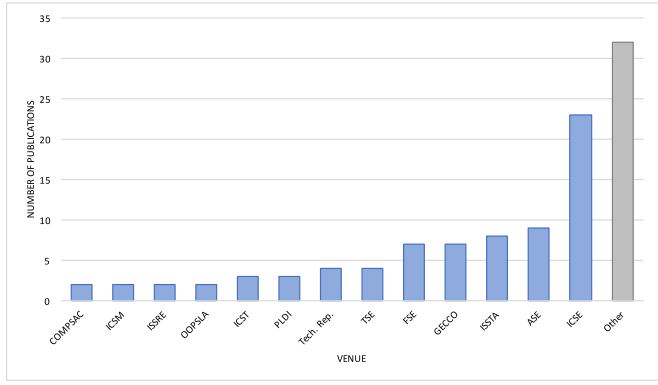

Fig. 2. Publications per venue.

interest of many researchers who produced a number of approaches for repairing different classes of faults under different conditions and hypotheses. Important results have been already achieved, but at the same time these results revealed the existence of relevant challenges that have to be faced.

The contributions in the area have been preliminary analyzed by Khalilian et al. [32], who compared three repair techniques, and Monperrus et al. [33], [34] who discussed repair and self-healing techniques focusing on the oracle problem.

This paper contributes to the research on program repair techniques by providing a comprehensive review of the available approaches, critically analyzing the capabilities of repairing techniques, surveying the results achieved so far, and identifying the key open challenges that should be faced by the research community in the future.

The paper is organized as follows. Section 2 describes the procedure we followed to select the papers for this survey and presents statistics about the selected papers. Section 3 introduces software repair solutions contrasting them with software self-healing solutions. Section 4 introduces a running example, consisting of multiple faulty versions of a same program that are used throughout the paper to illustrate the repairing techniques. Section 5 presents the localization techniques that have been exploited as part of the program repair solutions. Section 6 presents the algorithms for generating fixes. Section 7 extends the discussion to techniques recommending fixes that can be turned into an actual fix with small manual effort. Section 8 presents the main empirical findings that have been obtained so far. Section 9 discusses open challenges and future research directions. Section 10 provides final remarks.

2 Paper Selection

We selected the papers for the survey by searching in the ACM Digital Library, the IEEE Explorer Digital Library, and the Google Scholar repository for the following terms commonly used in software repair papers: ''program repair'', ''software repair'', ''automatic patch generation'', ''automatic fix generation'', ''generate and validate'', and ''semantics driven repair''. We performed the paper selection at the beginning of January 2017. For each query, we extracted the first 200 references sorted by relevance. Since a few queries returned less than 200 references, we ended up with a total of 3,262 references to be analyzed. After removing duplicated entries, we obtained a population of 2,573 references.

We manually analyzed all these references by checking the abstract, the introduction, and the conclusion sections, and scanning the rest of the paper, to quickly eliminate papers clearly unrelated to the topic of this survey. At the end of this process, we obtained a total of 107 possibly relevant papers. We carefully read these papers to determine their relevance according to the conceptual framework discussed in Section 3. This produced a total of 85 relevant papers. To be as much comprehensive as possible, we included the relevant papers that we missed with our searches but were cited in the papers we selected. This activity produced 23 additional references that have been included in the survey [26], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56] for a total of 108 selected papers. The resulting material has been used to produce this survey.

We analyzed the distribution of the selected papers per year and per publication venue. Fig. 1 shows the number of papers published on automatic program repair per year. After some early papers on this subject, we can notice a growing number of papers published every year starting from 2005. If we compare the number of papers published in the early 2000s to the papers published in the last few years, we can notice an order of magnitude increment in the number of published papers. This demonstrates a growing interesting of the research community on this subject.

Fig. 2 shows the number of papers published in each publication venue. We report explicitly every venue with at least 2 publications, while we collapse all the venues with a single publication in the column *Other*. We can notice a prevalence of software engineering conferences, with ICSE being by far the venue including the highest number of papers. Conferences related to genetic computation (e.g., GECCO) and programming languages (e.g., PLDI and OOPSLA) are also present. This is a consequence of the nature of the approaches to automatic program repair, which are often based on search and synthesis techniques, and can target a number of languages and environments.

Finally, we checked the distribution of papers between journals and conferences, and found that only 8.3 percent of the publications appeared in journals, while 91.7 percent appeared in other venues, such as conferences and workshops.

3 Software Repair

The problem of automatically generating a fix for a faulty program relates to the more general problem of automatically generating a program that obeys to a given specification.

This problem is an instance of *automatic programming*, which is the process operated by a machine of translating a specification about a task into a machine executable program for doing that task [57], [58]. Although the problem of automatically synthesizing software programs has been considered from the early ages of computer science and is still an active area of research [59], [60], [61], the problem of automatically repairing faulty programs have been considered only recently.

A few approaches *pioneered* the area of automatic program repairing. Stumptner and Wotawa introduced the idea of defining fault modes to capture frequent faults in VHDL programs [46], [62]. Fault modes have been used to automatically check if a program misbehaves consistently with any known fault and attempt to automatically repair the program with strategies depending on the recognized fault mode. Staber et al. investigated how to automatically replace the components of a system to make it satisfy a given Linear Temporal Logic (LTL) specification [47]. Interestingly, they defined the repair process as a game between the system, which has to be repaired, and the environment, which acts against the system to make the repair fail [43], [47]. Weimer investigated how to repair a program that violates a safety policy specification that specifies the allowed paths in the control flow graph of the program under repair. His approach finds the smallest set of modifications to be performed on the method calls produced by a program to make it satisfy the specification [31]. Other researchers extended model checking with artificial intelligence techniques [48] or template-based strategies [49] to repair programs.

Research in software repair has started proliferating after the publication of the *seminal* work by Arcuri et al. [35], [40], [63] and Weimer et al. [20], [64], [65], who have been the first ones to successfully exploit search-based algorithms to generate fixes.

In the rest of this section, we introduce the basic concepts relevant to the software repair area, and present a conceptual framework that we use to show how the software repair technology works in general, and how it relates to the software self-healing technology.

3.1 Basic Concepts

Two well-known approaches for automatically dealing with program failures are the *software healing* and *software repairing* technologies. Although similar in principle, they have different purposes and use different solutions to address faults.

Software healing solutions *detect software failures in-the-field* and respond to them by making the necessary adjustments to *restore the normal operation* of a system [66], [67], [68]. These adjustments are not deployed at the source code level, but rather *applied at runtime on the deployed application* to prevent or mitigate failures. Multiple occurrences of similar failures on the same instance of the software may trigger the same healing process multiple times.

Software repairing solutions *detect software failures*, *localize* where a fix could be applied, and make the necessary adjustments to *fix the fault*, and thus prevent further failures caused by the same fault [20], [21], [22]. The adjustments are generated *in-house*, such as at testing and design time, and *deployed at the source code level*.

Software healing and repairing may or may not involve human intervention. In *automatic* software healing and repairing, the human only supervises the process. While, when the human is completely out of the loop, these techniques are respectively called software *self-healing* and software *self-repairing*.

Both healing and repairing techniques react to failures by executing some operations. We distinguish between *healing operations*, which are the operations applied at runtime to turn a failed or a failing execution into a successful execution, and *repair operations*, which are the operations performed on the program source code to remove the fault that caused a failure.

Software healing and repair techniques may generate either workarounds or fixes. In this context, a *workaround* is a *temporary* solution to a software bug [69], while a *fix* (or a *patch*) is a *permanent* solution to a software bug [70]. Workarounds are not designed to implement optimal solutions. They rather consist of solutions that can be quickly generated and deployed, while waiting for a permanent fix from the developers. Workarounds can be applied at multiple levels and on different artifacts. For instance, a workaround may alter the program code, but can also modify the architecture of a system or change its configuration, or can simply alter a specific execution, for example letting users to interrupt unwanted infinite execution loops [71]. Contrarily, fixes always alter the program source code and are expected to implement solutions of the same quality of the fixes produced by the developers when debugging and fixing software faults.

3.2 Conceptual Framework

In this section, we present a conceptual framework that covers both healing and repairing solutions, and that illustrates the commonalities and differences between these two approaches.

Although they both deal with failures, software healing and software repairing solutions have different objectives. Software healing solutions are concerned with granting software availability despite failure occurrences [66], [67], [68]. They usually execute healing operations to mask failures [69], [72], [73], or minimize their impact [74], [75]. Identifying and permanently removing the fault that caused a failure is not a primary objective of these techniques. Software repairing solutions instead are concerned with fixing faults, and do not necessarily prevent failures [20], [21], [22]. Indeed failures are tolerated and information extracted during failing executions is even exploited by software repairing solutions to better identify and fix faults. Due to these distinguishing characteristics, software healing solutions are used in-the-field to improve the reliability and the resilience of software systems, and software repairing solutions are used in-house to assist developers in bug fixing tasks.

Fig. 3 graphically illustrates the activities involved in the healing and repairing processes, represented on the left and right side of the figure, respectively. Both processes start from a faulty program that fails deviating from its intended

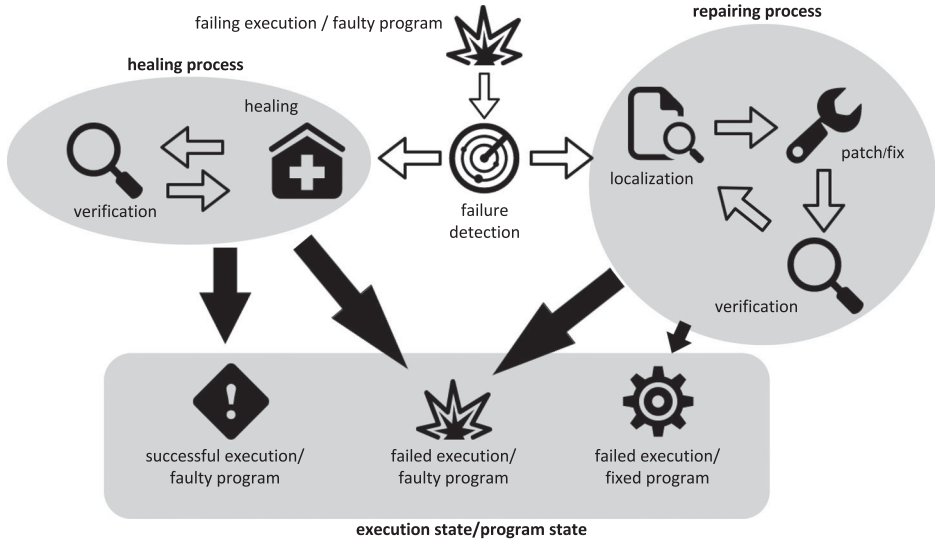

Fig. 3. Healing and repairing processes.

behavior, at least for some executions. The *failure detection* activity, which is common to both processes, is responsible of classifying executions as either failures or failure-free executions.

Failure detection might work differently depending on the process that must be activated. *When serving the healing process*, failure detection typically detects the early symptoms of program failures to timely trigger the healing process and prevent severe problems, such as system crashes and loss of data. *When serving the repairing process*, failure detection typically partitions a set of observations into failure and failure-free executions. These executions are then processed to identify and fix the faults that originated them. In contrast to the healing process, the repairing process does not prevent failures, but exploits the information collected during complete (failing) runs to identify faults.

The way failure detection serves the healing and repairing processes also influences the implementation of the failure detection component. When used to trigger healing operations, it is typically implemented as a runtime monitor that checks the behavior of the application [76], [77]. When used as part of a repairing process, it is typically implemented as a program oracle [78], for instance as assertions embedded in a set of test cases [79].

The *healing process* shown on the left side of Fig. 3 is composed of two steps that might be executed iteratively. The *healing* step consists of executing a healing operation that can prevent or mitigate a failure that has been detected. The *verification* step checks if the application is running as expected after the healing operation has completed. Not all the healing techniques include a verification step, sometime healing operations are applied simply assuming they can only positively or neutrally affect the system. If the verification step detects that the healing operation has failed, the healing step might be repeated, for instance trying a different healing operation and verifying the system again, until the failure has been healed, or no further actions are possible [80], [81]. The healing process usually requires fast healing and verification operations because they are both executed while the application is running in-the-field.

Since the primary objective of healing techniques is turning failing executions into successful executions, regardless of the fault that originated the problem, the healing process might produce two possible outcomes: (i) *successful executions/faulty program*, that is, the executions have been healed but the program is still faulty, (ii) *failed executions/faulty program*, that is, the executions have not been healed and the program is still faulty. In some cases, a successful workaround might be deployed persistently on the running application to automatically heal further occurrences of the same failure, although it can seldom cope with slightly different failures triggered by the same fault.

The *repairing process* shown on the right side of Fig. 3 is composed of three steps. The *localization* step identifies the locations where a fix could be applied to (note that the faulty statements are not always the only good locations for fixes). The *patch/fix* step generates fixes that modify the software in the code locations returned by the localization step. The *verification* step checks if the synthesized fix has actually repaired the software. The fix and verification steps might be iterated several times for multiple locations until the fault has been fixed, until no further fixes can be generated, or until the time allocated to the repair process has been exhausted.

If we compare the repair process to the healing process, it is possible to notice that the repair process includes an extra step. While a healing process reacts directly to failures, the repair process needs to first identify locations suitable for fixes.

In contrast to the healing process, which aims to prevent failures, software repairing techniques exploit the observations collected with multiple runs, including both failures and failure-free executions, to generate fixes. For instance, many repair techniques generate fixes by using the information obtained from the execution of large test suites that include many passing and failing tests.

The repair process might produce two possible outcomes: (i) *failed executions/fixed program*, that is, the executions have failed but the program has been fixed, thus the fixed fault will not produce further failures, (ii) *failed*

executions/faulty program, that is, the executions have failed and the program is still faulty. The first outcome corresponds to a successful repair, while the second outcome corresponds to an unsuccessful repair.

In the following sections, we describe automatic program repair solutions[1], focusing on the strategies for localization, fixing, and verification. The description is guided by several sample faulty programs that are introduced in the next section. We finally extend the discussion to nearly automatic approaches that can recommend fixes but require the intervention of the developer to finalize and incorporate the fixes in the program under repair.

4 Example Programs

In this section, we introduce the running examples that we use throughout the paper to illustrate program repair techniques. In particular, we use five faulty variants of a program that computes the greatest common divisor using the subtract-based version of the Euclid's algorithm. The correct version of the program is shown in Algorithm 1 and has been also used in Weimer et al.'s paper [20] to present the GenProg program repair technique. Algorithms from 2 to 6 include a fault highlighted with a grey background.

Algorithm 1. gcd(int a, int b)

```
1: if(a==0){
2:    printf("%", b);
3:    exit(0);}
4: while(b != 0){
5:    if(a > b)
6:       a = a - b;
7:    else
8:       b = b - a;}
9: printf("%d", a);
10: exit(0);
```

In Algorithm 2, we modified the `if` condition evaluated at line 1 by introducing an extra clause. This fault causes the program to loop indefinitely on statement 8 every time `a` is `0` and `b` is greater than `0`.

Algorithm 2. gcdWrongIf1(int a, int b)

```
1: if( a == 0 && b == 0) { //a==0
2:    printf("%d", b);
3:    exit(0);}
4: while(b != 0){
5:    if(a >b)
6:       a = a - b;
7:    else
8:       b = b - a;}
9: printf("%d", a);
10: exit(0);
```

In Algorithm 3, we modified the `while` condition at line 4 from `b!=0` to `a!=0`. This fault causes the program to loop indefinitely at line 6 when the input value of `a` is greater than `0` and the input value of `b` is equal to `0`.

1. We omit the term *automatic* when obvious from the context.

Algorithm 3. gcdWrongWhile(int a, int b)

```
1: if(a==0){
2:    printf("%d", b);
3:    exit(0);}
4: while(a != 0){ //b!=0
5:    if(a > b)
6:       a = a - b;
7:    else
8:       b = b - a;}
9: printf("%d", a);
10: exit(0);
```

In Algorithm 4, we removed the `exit(0)` statement from the `if` block at line 1. This fault produces the same failure than the one considered in Algorithm 2, that is, when `a` is `0` and `b` is greater than `0` the program loops indefinitely on statement 7. This same fault has been used in [20].

Algorithm 4. gcdNoExit(int a, int b)

```
1: if(a==0){
2:    printf("%d", b);} //missing exit(0);
3: while(b != 0){
4:    if(a >b)
5:       a = a - b;
6:    else
7:       b = b - a;}
8: printf("%d", a);
9: exit(0);
```

In Algorithm 5, we modified the `if` condition at line 1 by changing the clause from `a=0` to `a<b`. This fault may cause the program to produce a wrong output when `a<b` and `a!=0`. In that case, the execution enters the first `if` statement and prints the value of variable `a`, which might not be the gcd of `a` and `b`.

Algorithm 5. gcdWrongIf2(int a, int b)

```
1: if(a < b){ //a==0
2:    printf("%d", b);
3:    exit(0);}
4: while(b != 0){
5:    if(a > b)
6:       a = a - b;
7:    else
8:       b = b - a;}
9: printf("%d", a);
10: exit(0);
```

In Algorithm 6, we modified the `print` statement at line 2, which prints the value of the wrong variable. The program outputs the wrong value every time `a` is `0` but `b! = 0`.

When relevant, this set of faulty programs is used to illustrate the capabilities and complementarities of the program repair algorithms surveyed in the next sections.

5 Localization

In this section, we discuss the algorithms used by program repairing techniques to identify the program locations where the fixes should be applied to. This is an important step of the program repair process because the success of a

repair may depend not only on the fix itself but also on the location of the fix.

Algorithm 6. gcdWrongPrint(int a, int b)

```
 1: if(a==0){
 2:     printf("%d ", a); //printf("%d", b);
 3:   exit(0);}
 4: while(b != 0){
 5:   if(a >b)
 6:     a = a - b;
 7:   else
 8:     b = b - a;}
 9: printf("%d", a);
10: exit(0);
```

We distinguish two main classes of approaches for the choice of the location: *fault localization* and *fix locus localization*. *Fault localization* techniques aim at identifying the faulty statements. In this case, the repair is expected to turn the discovered faulty statements into correct statements. For example, in the case of Algorithm 2, a fault localization technique would aim at identifying that the `if` condition at line 1 is the right place for the generation of a fix.

Fix locus localization techniques (also known as fix localization techniques) aim at detecting the statements where the effect of the fault can be observed and eliminated. These statements are not necessarily the statements with the fault, but they are potentially modifiable to let the program behave correctly. In fact, a faulty behavior may often be fixed by compensating the effect of the fault, rather than removing the fault. For instance, instead of fixing the condition at line 1 of Algorithm 2, it would be possible to fix that same program by adding another `if` condition that prints the value of `b` when `a` is equals to `0` between lines 3 and 4. Thus, a fix locus localization technique may determine line 4 as a suitable place for the generation of a fix that may compensate the fault at line 1.

Although the approaches in the two classes have different purposes, in the practice a fault localization technique may end up identifying a fix location, and vice versa a fix locus localization technique may end up identifying the location of the fault.

Both fault and fix locus localization strategies are often defined ad-hoc, coherently with the capabilities and the requirements of the corresponding program repair technique. For instance, AFix [82] can repair the atomicity violation faults specifically localized with CTrigger [83], which combines trace analysis and execution perturbation to extract information about the failure and the faulty code regions.

When necessary to understand the approach, we describe the fault localization strategy together with the repairing techniques. In this section, we discuss classes of localization approaches that have been exploited by multiple techniques. We refer to these approaches when relevant throughout the paper.

5.1 Fault Localization

Program repairing techniques extensively use Spectrum-Based Fault Localization (SBFL) [84] as general mechanism to localize the statements that are likely to be faulty.

SBFL exploits information about the behavior of the software collected during testing to identify the program elements that are likely to be faulty. In particular, SBFL produces a list of program elements ranked according to their likelihood of being faulty based on the analysis of the program entities covered by passing and failing tests [6], [85], [86]. The general intuition is that the program elements executed by many failing test cases and few passing test cases are likely to be faulty, and vice versa the program elements executed by many passing test cases and few, or no, failing tests are likely to be correct. Program entities of different kind and granularity might be considered for the localization, such as statements, branches, and paths. Fault localization and program repair usually work at the same granularity level, that is, if the program repairing technique works at the level of program statements, the SBFL algorithm used to localize the fault also works at the statement level.

We analyzed the papers presenting the techniques that exploit SBFL and we discovered that four fault localization solutions have been more frequently used than others so far: the fault localization algorithm defined in GenProg (32 percent of the papers that use SBFL) [87], [88], Tarantula [6] (18 percent of the papers that use SBFL), Ochiai (18 percent of the papers that use SBFL) [89], and Jaccard [90] (5 percent of the papers that use SBFL). Thus, to illustrate how SBFL works, we describe how these four techniques perform with the faulty program shown in Algorithm 6.

SBFL requires the availability of a test suite with passing and failing tests. The failing tests reveal the fault that must be localized and repaired. In the example, we assume a test suite with four passing test cases and one failing test case is available. Fig. 4 shows the faulty program (on the left), the test cases (as headers of the columns, the pair of numbers indicates the values of the program inputs `a` and `b`, respectively), the statements executed by each test case (each executed statement is indicated with an `X`), the test outcome (either *P* for pass or *F* for fail), and the suspiciousness score returned by Tarantula, Ochiai, GenProg's fault localization, and Jaccard. The score is a value between 0 (least suspiciousness) and 1 (highest suspiciousness) and is used to rank the program statements. We use a grey background to highlight the statements with the highest score according to Tarantula, Ochiai, and Jaccard. Note that the faulty print statement is in the set of the most suspicious statements for all the techniques. In the following, we illustrate how the four techniques work.

We use $failed(s)$ to indicate the number of failed test cases that execute the statement s, $totalfailed$ to indicate the total number of failed test cases in the test suite, $passed(s)$ to indicate the number of successful test cases that execute the statement s, $totalpassed$ to indicate the total number of successful test cases in the test suite, and $execute(s)$ to indicate the total number of test cases that execute s.

Tarantula computes the suspiciousness of a statement as the ratio between the rate of failed test cases that execute that statement, and the sum between the rate of passed and failed test cases that execute that same statement. More formally, the suspiciousness $suspT$ of a statement s is

$$suspT(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}} \quad (1)$$

	(0,0)	(0,10)	(10,0)	(10,15)	(15,10)	Tarantula	Ochiai	GenProg	Jaccard
gcdWrongPrint(int a, int b) {									
1: if (a == 0){	X	X	X	X	X	0.5	0.4	0.1	0.2
2: printf("%d", a);	X	X				0.8	0.7	0.1	0.5
3: exit(0);}	X	X				0.8	0.7	0.1	0.5
4: while (b != 0){			X	X	X	0	0	0	0
5: if (a >b){				X	X	0	0	0	0
6: a = a - b;				X	X	0	0	0	0
7: }else{				X	X	0	0	0	0
8: b = b - a;}				X	X	0	0	0	0
9: printf("%d", a);			X	X	X	0	0	0	0
10: exit(0);}			X	X	X	0	0	0	0
test outcome	**P**	**F**	**P**	**P**	**P**				

Fig. 4. SBFL with Tarantula, Ochiai, GenProg, and Jaccard.

Ochiai [89] computes the suspiciousness of a statement as the ratio between the number of failed tests that execute s and the square radix of the product between the total number of failed tests and the total number of statements that execute s. More formally, given a statement s, its suspiciousness $suspO$ is

$$suspO(s) = \frac{failed(s)}{\sqrt{totalfailed \times (failed(s) + passed(s))}} \quad (2)$$

GenProg [87] assigns to each statement one of three possible scores representing the case the statement is never executed by failed test cases, is only executed by failed test cases, or is executed by both failed and passing test cases. More formally, the suspiciousness *suspG* of a statement s is:

$$suspG(s) = \begin{cases} 0 & failed(s) = 0 \\ 1.0 & passed(s) = 0 \wedge failed(s) > 0 \\ 0.1 & otherwise \end{cases} \quad (3)$$

Jaccard [90] computes the suspiciousness of a statement s as the ratio between the number of failed tests that execute s and the total number of tests that execute s summed to the failing tests that do not execute s. More formally, given a statement s, its suspiciousness $suspJ$ is

$$suspJ(s) = \frac{failed(s)}{execute(s) + (totalFailed - failed(s))} \quad (4)$$

Note that the more failed tests execute a statement, the higher the Tarantula, Ochiai, and Jaccard scores are. Vice versa the more passed test cases execute a statement, the lower the Tarantula, Ochiai, and Jaccard scores are. GenProg's fault localization follows a similar pattern, although based on three possible values only. In particular, all the techniques assign a score of 0 when no failed tests execute a statement, and a score of 1 when all and only the failed tests execute a statement. In the example shown in Fig. 4, the lowest suspiciousness is assigned to the statements in the `while` loop, since no failed test case executes them. The `if` statement has a non-zero suspiciousness score because it is executed by all the test cases, including the failed ones. Finally, the most suspicious statements are the ones in the `then` branch of the `if` condition (including the condition itself for GenProg's fault localization), which includes the statement with the fault.

In the example, a program repair solution that uses one of these fault localization techniques would focus on the first three statements to synthesize a proper repair. Note that in the example the most suspicious statements include the faulty statement, although this is not guaranteed to happen in general, that is, the faulty statement might be ranked at lower positions if a different set of test cases is provided. The impact of these techniques on the effectiveness of the repair algorithms has been preliminary investigated by Qi et al. [91] and Jaccard has been observed to best support automatic program repair techniques.

5.2 Fix Locus Localization

Fix locus localization techniques aim to identify the program locations suitable for the synthesis of fixes, regardless of the location of the faults. The idea is that a program location where the effect of a fault could be recognized could also be exploited to compensate its effect. In the following we describe model-based fix locus localization, which can identify statements that use objects illegally in object-oriented software [25], and angelic fix localization, which can localize the statements relevant to missing or faulty *if* conditions faults [23].

5.2.1 Model-Based Fix Locus Localization

Model-based fix locus localization has been experienced in the context of program repair techniques that can repair faults that cause the incorrect usage of class interfaces [25], [92], [93], [94].

Model-based fix locus localization works by first running the passing test cases to collect runtime data and then mining models that represent how classes are used by the program during correct runs. The mined models consist of finite state machines that represent the object life-cycle, that is, how objects are created, when the methods of these objects can be invoked, and when the fields of these objects can be written and read. The mined models are then checked dynamically while the failing test cases are executed. A failed execution that violates a mined model indicates the presence of objects that are incorrectly used by the program, such as code that writes data on a file that has been opened in read-only mode. The program statements that incorrectly use objects are the locations targeted by the fix generation process. Note that these statements, although

not necessarily faulty, are clearly appropriate places for enforcing the legal use of objects interfaces.

Although not always experienced in the context of software repair, the idea of mining models from correct executions to successively check failing executions and identify the code elements likely responsible for the failure is shared with several others failure analyis techniques [14], [15].

5.2.2 Angelic Fix Localization

Angelic fix localization targets faults that can be repaired by modifying the *if* conditions in a program [23]. The idea is to first localize the suspicious conditions according to their suspiciousness score, and then identify which of these conditions might be wrong by systematically forcing the failing test cases to take alternative branches at decision points, regardless of the outcome of the evaluated condition. For instance, all the inputs with `a` equals to `0` and `b>0` cause Algorithm 2 to fail because the condition at line 1 evaluates to false and the execution does not take the `then` branch. Angelic fix localization would check what happens when the `then` branch is taken by the execution, although the condition evaluates to false. Taking the `then` branch causes the execution to pass, in fact the program would terminate after printing `b`, which is the right result for all the inputs with `a` equals to `0` and `b>0`. These conditional values that make test cases pass are called *angelic values*. All the decision points that might be used to turn failing test cases into successful test cases according to this simple strategy are selected and returned as possible targets of an automatic program repairing solution designed to address the conditions in a program.

The above approach works in a slightly different way when attempting to detect missing *if* conditions. Angelic fix localization checks if failing test cases can be turned into passing test cases by skipping the execution of single (simple or compound) statements. It investigates the effect of every suspicious statement and returns those statements whose execution should be skipped during failing test cases as targets for the synthesis of fixes. In this case, a fix should consist in the addition of new *if* statements that properly control the execution of the statements that must be skipped. The synthesis of the appropriate conditions in the *if* statements is not part of the fault localization approach, but of the fixing process.

An improved version of angelic fix localization [95] extends the original technique with the capability to consider any program expression (instead of only expressions in conditional branches) when extracting angelic values. This is done by first identifying the suspicious program locations and the corresponding expressions with SBFL, and then running a customized symbolic execution that determines the angelic values while replacing the suspicious expressions with symbols.

Note that the fix locus locations returned by angelic fix localization are not necessarily the fault locations. The locations identified by angelic fix localization are places where the flow of the executions could be opportunistically altered, by modifying or adding conditions, to mask or compensate the effect of a fault, and do not necessarily correspond to places with faulty statements.

6 Fix Generation

This section discusses the algorithms for the automatic generation of program fixes and the corresponding validation strategies. All the algorithms *approximate* the problem of generating fixes with a problem P_{repair} to be solved. This implies that a solution for P_{repair} is likely to solve the original problem of producing a fix, but there is no guarantee this will happen in practice. For this reason, a solution s_{repair} to P_{repair} is called a *plausible solution*.

We distinguish two main classes of approaches depending on the way P_{repair} is defined and addressed: *generate-and-validate* and *semantics-driven* approaches.

Generate-and-validate approaches produce fixes by defining and exploring a space S_{repair} of the potential solutions to P_{repair} (note that S_{repair} may contain both elements that solve and do not solve P_{repair}).

Semantics-driven approaches (also known as *correct-by-construction* approaches) encode the problem P_{repair} formally, either explicitly or implicitly, and once they find a solution, the solution is guaranteed to solve P_{repair}.

Regardless of the class of approaches that is used, any solution s_{repair} is always reported to the developers who cross-check its quality and correctness, further elaborating it to obtain a fully satisfactory and correct fix, if necessary. Regardless of the need of editing an automatically generated fix, the availability of an automatic fix may already help developers understanding the fault in the code and simplify the implementation of the actual fix.

Repairing techniques might be designed to be either *general* or *fault-specific*.

General techniques are not designed to target a specific class of faults and can potentially repair any class of faults in a program.

Fault-specific techniques are designed to address some classes of faults only, such as wrong conditions in conditional statements and buffer overflows.

In principle fault-specific techniques should balance their narrower scope with a higher efficiency of the repair process.

In the rest of this section, we first present generate-and-validate techniques and then semantics-driven techniques. We first introduce the *key concepts* that characterize each class of approaches and we provide a *roadmap* that organizes contributions according to several dimensions, then we discuss the individual approaches, and finally exemplify them using the example cases introduced in Section 4.

6.1 Generate-and-Validate

Generate-and-validate approaches execute an iterative process as shown in Fig. 5. The process consists of two main activities, the *generate* activity, which produces candidate solutions to the P_{repair} problem, and the *validate* activity, which checks the correctness of the generated solutions.

The *generate* activity uses a set of *change operators* to modify the original program P and produces k new programs that are added to the set of the *candidate solutions*. The modifications are performed with higher probability on the suspicious locations identified by localization techniques. The change operators might be of different nature.

Atomic change operators modify P in a single point.

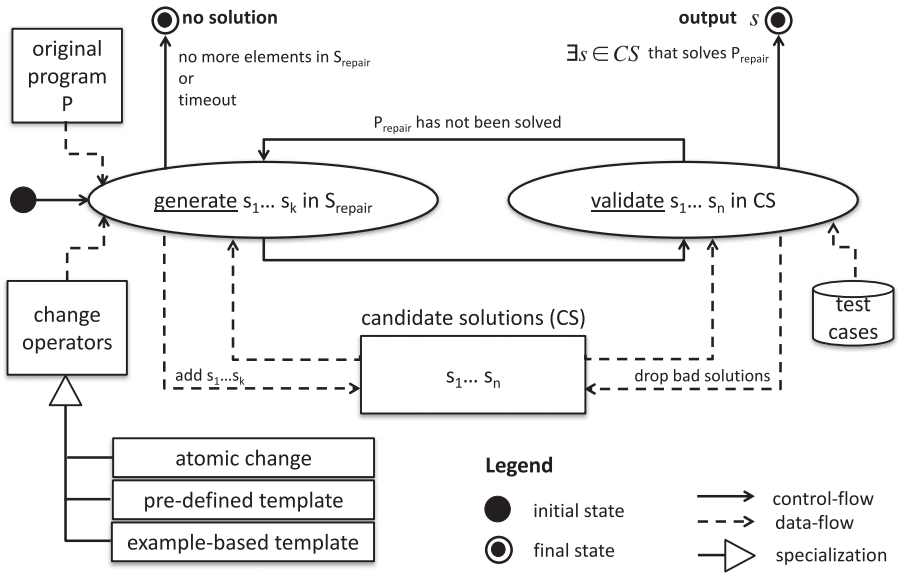

Fig. 5. Generate-and-validate repair process.

Pre-defined template operators change P according to potentially complex pre-defined patterns.

Example-based template operators work the same than pre-defined templates, but the templates are extracted, either manually or automatically, from historical data (e.g., a versioning system).

Some techniques do not only apply the change operators to P only, but also to the candidate solutions [22], [27], [36], [38], [63], [64], [96], [97]. In this way, the modified programs may incrementally accumulate changes produced by the application of multiple change operators. The overall set of all the candidate solutions that can be produced by a technique represents its search space S_{repair}. When every possible element in S_{repair} has been considered or the amount of time allocated to the repairing process has expired, the process ends producing no solution to P_{repair}.

The *validate* activity checks for the correctness of the candidate solutions. So far, most of the generate-and-validate techniques establish the correctness of the solutions by running the available *test cases*, that is, if a program s in the set of the candidate solutions passes all the available test cases, the program s is returned to the developer as a possible fix for P. As outcome of the validation, some or all the elements in the set of the candidate solutions might be discarded, for instance the candidate solutions that are largely unsatisfactory because they fail many tests are usually discarded.

The generate and validate activities might be executed according to two main strategies: *search-based* and *brute-force*. These strategies differ on the way the change operators and the candidate solutions are handled.

Search-based strategies apply change operators randomly or guided by a heuristic or meta-heuristic search algorithm.

Brute-force strategies systematically produce every possible change that can be obtained within certain bounds with the considered change operators and localization algorithm.

Note that in the literature the label ''search-based'' has been used both to specifically identify the techniques that use search algorithms and to indicate the whole class of generate-and-validate techniques. In this paper, we use the term search-based *uniquely* to indicate the subclass of generate-and-validate techniques that use search algorithms.

Tables 1 and 2 report the techniques that implement the generate-and-validate approach (column *Techniques*). Table 1 includes the techniques that use *atomic change operators* to generate fixes, while Table 2 includes the techniques that use *template-based change operators*. The techniques are classified based on the search strategies used to generate the candidate solutions (column *Strategy*), the type of fault that is addressed (column *Fault model*), and the code entities that are modified in the attempt to repair faults (column *Change model*). Note that we use the label *General* in the column Fault model to indicate repair techniques that are not designed to address a specific class of faults, while the other labels indicate specific classes of faults. Finally, column *Section* indicates the section of the paper that describes the techniques listed in the corresponding rows. In the rest of this section, we discuss the cases reported in the table.

6.1.1 Atomic Change Operators

An atomic change operator modifies a program in a single place of its Abstract Syntax Tree (AST), for instance inserting, deleting, or modifying a statement or even a single operator of an expression. It is the simplest class of change operator because it does not require any analysis of the program but can be implemented by only analyzing the program location that should be changed.

Several techniques use these simple change operators because they can be efficiently applied and combined together to obtain many variants of the program under repair, to possibly find a fix. On the contrary, change operators that require a more expensive analysis of the program under repair may slow down the search process, although they might be more effective in some cases. In the following, we discuss the generate-and-validate techniques that use atomic change operators, grouping the techniques working on a similar way.

General Search-Based Techniques. The search-based program repair techniques designed so far (see row *search-based* in Table 1) implement a randomized search process

TABLE 1
Generate-and-Validate Techniques—*Atomic Change Operators*

Strategy	Fault model	Change model	Techniques	Section
search-based	general	AST reuse/insert/delete	GenProg, Marriagent, RSRepair, SCRepair	Sec. 6.1.1.1
		AST modifications	JAFF	
		operator replacement, variable name replacement	pyEDB, MUT-APR, CASC	
brute-force	general	operator replacement, condition negation	Debroy and Wong	Sec. 6.1.1.2
		method call insertion/deletion	PACHIKA	
		functionality deletion	KALI	
		AST reuse/insert/delete	AE	

designed to potentially cope with any kind of fault in the code. The techniques in this category [20], [27], [29], [36], [50], [96], [98], [99] use slightly different algorithms and different heuristics to maximize the effectiveness of a set of atomic change operators defined at the level of the AST of the program, that is, each operator manipulates a single element of the AST typically deleting, modifying, and inserting nodes in the AST, but also modifying the individual operators and variables used in the statements.

In particular, GenProg, Marriagent, RSRepair, and SCRepair use the same three atomic change operators: deleting a statement of the AST, inserting a statement copied from a random location of the AST, modifying an element of the AST with an element copied from a random location of the AST. This set of operators constitutes a quite general change model that can arbitrarily modify a program, as long as the modified program includes a sufficient variety of statements that can be copied from one location to another. JAFF extends this set of atomic changes with operators that can manipulate subtrees of the AST and operators that can randomly generate new statements in the program, further expanding the set of changes that can be produced at each step of the process.

Note that these change operators are general and potentially useful to repair any fault in the program, as long as the statements that must be used to produce the fix are still contained elsewhere in the program (this hypothesis is known as the *plastic surgery hypothesis* [100]). For instance, the faulty algorithm shown in Algorithm 4 could be fixed by these techniques by copying the statement `exit(0);` from line 9 of the program to line 2 of the same program. Of course, if the program does not include the statements necessary for the fix, these techniques cannot repair the faults, such as for Algorithm 2 that requires changing the `if` condition with a condition that does not occur elsewhere in the program. JAFF represents a small exception to this rule, because it includes the capability of randomly generating new statements, which makes it potentially useful with a larger number of faults, although the probability to produce the statements necessary for a fix randomly is extremely small.

pyEDB, MUT-APR, and CASC also aim to potentially fix any kind of fault, however they choose to adopt a rather different change model. Instead of using change operators that can produce a large set of different changes, they use a small set of focused change operators. In particular, these techniques focus on the use of change operators that can modify the operators and the variables used in the target program.

In the following, we discuss more in details the individual techniques.

GenProg [20], [64], [65], [87], [88], [101] is a repair technique that uses *genetic programming* to guide the generate and validate activities. At every iteration, the location where an atomic change is applied to is determined randomly, according to a probability distribution that matches the suspiciousness of the statements computed with spectrum-based fault localization algorithms, as described in Section 5.1. The intuition is that changes have a higher probability to affect a fault if the target statements well correlate to the failure.

When a non-empty set of candidate solutions is available, the generate activity, in addition to apply the atomic change

TABLE 2
Generate-and-Validate Techniques—*Template-Based Change Operators*

Template type	Strategy	Fault model	Change model	Techniques	Section
pre-defined	search-based	concurrency faults	synchronized region manipulation	ARC	Sec. 6.1.2.1
	brute-force	general	code transformation templates	AutoFix-E, AutoFix-E2	Sec. 6.1.2.2
			condition change, variable value change	SPR, Prophet	
		buffer overflow	buffer manipulation, function replacement	PASAN, AutoPAG	Sec. 6.1.2.3
example-based	search-based	general	code transformation templates, reuse of statements in the same application	History-driven repair	Sec. 6.1.3.1
			code transformation templates	PAR, Relifix	
	brute-force	general	code transformation templates	R2Fix	Sec. 6.1.3.2
		buffer overflow	insertion of code fragments from donor programs in the code under repair	CodePhage	Sec. 6.1.3.3

operators, also applies single-point crossover, which randomly selects two candidate solutions, randomly selects a point in the two solutions, and produces two new candidate solutions by juxtaposing the initial part of the first candidate solution to the final part of the second candidate solution, and vice versa.

Every candidate solution is validated running the available test suite. GenProg defines a fitness function that measures how good each program variant is based on the number of passing and failing test cases. In particular, the fitness of a program is defined as the weighted sum of the number of passing and failing test cases. Passing test cases are associated with a positive weight, while failing test cases are associated with a negative weight. The program variants with high fitness are preserved, while the program with low fitness (fitness below or equal to 0) are discarded.

The whole process is consistent with classic genetic programming based on mutation operators (i.e., the atomic change operators), selection (i.e., the capability to discard candidate solutions that fail many test cases), and crossover (i.e., the capability to mix the good solutions that survived to selection). The search space S_{repair} for GenProg consists of all the programs that can be obtained by applying an arbitrary number of atomic changes and crossovers.

The effectiveness of GenProg has been also studied in presence of some optimizations. For instance, when a long time is required to recompile and reinstall a program, weak recompilation techniques can be exploited to run the test cases on the candidate solutions that are incrementally generated, without having to recompile the whole program [102], [103]. Similarly, when GenProg is applied to x86 Assembly code and Java bytecode, the changed functions can be recompiled in the form of shared libraries, which are invoked instead of the original version of the functions in the program, thus saving the time necessary to recompile and reinstall the entire program [104] [105].

Marriagent [96] is a technique that works similarly to GenProg but uses a different crossover algorithm. Since selecting individuals for crossover randomly or based on fitness may cause the selection of similar individuals, which would likely produce similar offspring that only marginally improve the quality of the population of the candidate solutions, Marriagent selects individuals for crossover favoring diversity. In practice, it measures the diversity between a pair of programs taking into consideration the common and the total set of changes applied to each program with respect to the original program. The probability to select a pair for crossover depends on the diversity of the programs in the pair.

RSRepair [29] (formerly *TrpAutoRepair* [106]) is a repair technique that uses *random search* to guide the fix generation process. At each iteration, it determines the change operator that must be applied to the program to be repaired randomly, while the location where the change should be applied to is determined according to a probability distribution that depends on the suspiciousness of the statements.

Compared to GenProg, RSRepair does not use evolution, thus it does not apply atomic changes to candidate solutions and does not perform crossover operations. The assumption is that random search may perform well, and even better, than an evolutionary algorithm in the field of automatic program repair, where smoothened and progressive evolution might be hard to define. At each iteration, RSRepair simply generates a single candidate solution ($k = 1$ in Fig. 5), which is immediately validated and then discarded if it does not pass all the test cases. As a consequence any candidate solution always includes a single change with respect to the program under repair P. This leads to a search space S_{repair} consisting of all the programs that can be obtained by modifying a single statement of the program under repair. Thus fixes that require multiple changes applied to multiple locations cannot be produced with this algorithm.

In order to speed up the validation of a candidate solution, RSRepair *prioritizes* the available test cases increasing the probability that an incorrect candidate solution could be detected by one of the test cases that are executed earlier in the test suite [107]. The prioritization strategy assigns to each test case a priority based on the number of candidate solutions that has been able to discard. The intuition is that the test cases that have been effective in revealing changes that do not fix the program should be executed earlier than others [106].

SCRepair [98] extends RSRepair introducing a metric that computes the similarity between two code fragments from their AST. This metric is used to guide the selection of the new code that can replace the existing code during the mutation process. In particular, SCRepair looks for code that is not identical to the code that must be replaced, but similar enough to integrate well with the code around the change location. If code satisfying this criterion is present in multiple locations of the program, developers can define the types of changes that are more likely able to fix a program and thus should be preferred in the selection (e.g., changes that introduce *if* conditions).

JAFF [63], [99] is a technique that exploits a set of change operators working at both the level of the individual nodes and the level of the subtrees of the AST of the program to be repaired. The location of the change is still determined based on the suspiciousness of the statements, but JAFF increases the randomness of the selection of the change location by first selecting n statements at random, and then choosing the statement with the highest suspiciousness among the selected ones. These change operators can be used to generate candidate solutions according to three possible search algorithms: random search, hill climbing, and genetic programming.

pyEDB [27] uses genetic programming to generate and evolve the candidate solutions. Compared to GenProg, pyEDB introduces a novel way to represent the candidate solutions that must be mutated. While GenProg works with a full representation of the code in the candidate solutions, pyEDB represents a candidate solution by considering its delta with respect to the program that must be repaired. In particular, a candidate solution is identified by the set of changes that have been applied to the program under repair. This new representation is particularly convenient because it is compact and efficient to handle. In fact, the set of changes necessary to obtain a candidate solution is always small compared to the size of the whole program.

To obtain this representation, pyEDB exploits the existence of a small and focused set of change operators. When a program P must be repaired, pyEDB first generates two modification tables: the first table includes every possible

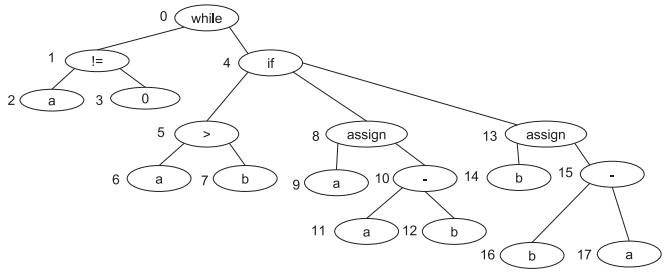

Fig. 6. AST of the while loop in Algorithm 3.

change to the relational operators that can be performed on P, and the second table includes every possible change to the variable names that can be performed on P. Changes must always lead to a valid program, thus variable names can only be replaced with other variable names that are in the scope of the modified instruction. pyEDB represents each change with a 32 bit string, where the first 4 bits identify the modification table, the next 20 bits identify the AST node that must be modified (i.e., they identify the row in the table), and the last 8 bits identify the specific change (i.e., they identify the specific change among the ones listed in the modification table for the selected AST node).

If we consider the while loop in Algorithm 3 whose AST is shown in Fig. 6, the corresponding modification tables are the following:

Modification table for arithmetic operators replacement

AST Node	*Possible changes*
1	$\{\geq, <, \leq, =, >\}$
5	$\{\geq, <, \leq, =, \neq\}$

Modification table for variable replacement

AST Node	*Possible changes*
2	$\{b\}$
6	$\{b\}$
7	$\{a\}$
...	...

The search space S_{repair} is defined by the programs that can be obtained by applying any combination of the changes reported in the modification tables. Thus any candidate solution can be represented as a set of 32bits strings.

The idea of focusing on the delta has been also explored in [102], where it has been exploited to optimize the compilation time by partially recompiling programs, dramatically reducing the cost of compilation.

pyEDB evolves the modification sets (i.e., the candidate solutions) using crossover and biasing the selection of the modification according to spectrum-based fault localization.

MUT-APR [36] is a repair technique very similar to GenProg: it uses spectrum-based fault localization to identify potentially faulty statements, and uses genetic programming to explore the search space of the possible program variants. The main difference is in the nature of the atomic change operators that are used to generate the search space. While GenProg is a general technique, MUT-APR exclusively addresses faults that can be repaired by modifying the operators used in a program, thus the change operators are limited to replacement of relational, arithmetic, bitwise, and shift operators.

CASC [50], [97], [108] uses genetic programming and atomic change operators mostly affecting programming operators and variable names. CASC elaborates the concept of co-evolution formerly introduced by Arcuri et al. [35], [40], that is, CASC evolves not only the candidate solutions but also the test cases. The intuition is that it is possible to produce better program variants and better test suites by rewarding the candidate solutions that pass the highest number of test cases and the test cases that discard the highest number of candidate solutions. The challenging aspect of this solution is to evolve the test suite without generating incorrect test cases that may steer the generation of the program variants toward faulty programs.

Sample Cases. Regarding the set of sample faults introduced in Section 4, GenProg, Marriagent, RSRepair, and SCRepair can only repair the fault reported in Algorithm 4. In fact, the fix consists of the insertion of the `exit(0);` statement in the `then` branch of the `if` condition at line 1, which can be obtained with the atomic change operator that inserts a statement already present in the program in another point of the program. In the case of Algorithm 4, the `exit(0);` statement is already present at line 9.

The other faults reported in Section 4 cannot be repaired with any of these four techniques because the correct statement is not present in the program. Of course, if the gcd routine would be part of a larger program, and somewhere else in the codebase the statements required to produce a fix occur, GenProg, Marriagent, RSRepair, and SCRepair might have a chance to repair these faults as well. In practice, GenProg, Marriagent, RSRepair, and SCRepair heavily depend on the assumption that the statements necessary to produce the fix are available somewhere in the program under repair.

For what it concerns JAFF, it might be able to repair any of the faults reported in Section 4, due to its ability to randomly generate new statements. However, to what extent this capability might be practically effective is not clear.

The capability of pyEDB, MUT-APR, and CASC to deal with the sample faults presented in Section 4 depends on the matching between the nature of the fault and the change model used by each technique. pyEDB, MUT-APR, and CASC can deal with faults that can be fixed by changing operators and variable names. In the sample cases, they can all fix the fault in Algorithm 5 by replacing < with ==. Since CASC and pyEDB can also change variable names, they can both fix the faults in Algorithms 3 and 6. While none of the three techniques can address the faults in Algorithms 2 and 4 because adding new statements and modifying complex *if* conditions are not in the scope of their change model, that is, the fixed program is not in the search space S_{repair} of these techniques.

General Brute-Force Techniques. Brute-force techniques search for a fix to a program under repair by exploring the search space *systematically*. In general, different techniques might use rather different strategies and different sets of atomic change operators to address different kinds of faults. The program repair techniques in this category use four types

of change operators (see row *brute-force* in Table 1): operator replacement and condition negation, method call insertion or deletion, functionality deletion, and AST manipulations.

The technique by *Debroy and Wong* [45] investigates the idea of exploiting the same operators used in mutation testing for automatic program repair. Thus, the atomic change operators considered by this technique are the replacement of an arithmetic, relational, logical, increment/decrement or assignment operator with another operator of the same kind, and the negation of *if*/*while* conditions.

The tecnique works by systematically changing the statements in the program under repair starting from the most suspicious one until reaching the least suspicious one, or until the time budget assigned to the repair process has expired. The search space of this technique only includes the programs that can be repaired by applying a single change to the program under repair.

PACHIKA [25] is a program repair technique that exploits specification mining techniques [109] to repair object-oriented software programs. The idea implemented in PACHIKA consists of observing and comparing the behavior of the software during the execution of passing and failing test cases to infer for each invoked method the preconditions that must be satisfied for its successful execution. The failing test cases must violate some of these preconditions, otherwise the fault cannot be repaired with this technique.

The fix strategy consists of modifying the program in the places identified with model-based fix locus localization (see Section 5.2.1) by adding or removing method calls in a way that may affect the violated preconditions. PACHIKA is limited to methods that do not require parameters. The removed methods are not physically deleted from the program, but are wrapped in an *if* block that skips their execution when the precondition does not hold.

Kali [110] is a technique that attempts to fix programs by removing, potentially unnecessary but harmful, functionalities. Kali pursues this strategy using spectrum-based fault localization to identify the 500 most suspicious statements of the program and systematically modifying these statements using a set of atomic change operators specifically designed to drop functionalities (e.g., setting an *if* condition to true or false, adding a return statement or removing statements). The search space considered by KALI consists of all the programs that can be obtained from the program under repair by applying one of the KALI's atomic change operators to one of the top 500 most suspicious statements.

Although KALI might succeed fixing faults in some cases, this repair solution has been primarily designed to experimentally investigate the issue of deriving plausible but incorrect fixes. In fact, programs passing test case execution can often be obtained by simply dropping functionalities, but these plausible fixes would be seldom considered correct fixes by a developer who generally aims to fix programs without losing functionalities. The challenge of generating correct fixes and not only plausible fixes is further elaborated in Section 9.1.

AE [111] considers the same search space S_{repair} considered by RSRepair, that is, the space of all the program variants that can be obtained by applying a single GenProg's change operator to a single statement of the program. The challenge is that this search space is often large and difficult to navigate. AE aims to reduce the size of the search space by exploiting semantic equivalent checking.

Semantic equivalence checking consists of the capability to detect sets of candidate solutions that are semantically equivalent although syntactically different. This capability allows to discard several candidates solutions without running the test cases, which is a typically expensive operation. AE can detect that two candidate solutions are semantically equivalent if they differ for any of those elements:

- *Syntactic equality*: programs that differ only for the presence of duplicated variable names or duplicated statements;
- *Dead code*: programs that differ only for the presence of some dead code;
- *Instruction scheduling*: programs that differ on the order of some adjacent instructions not referring to common resources, which can be reordered without affecting the semantics of the program.

AE is deterministic as long as the program under repair is deterministic because it exploits the ranking produced by SBFL to systematically mutate the statements in the program, from the most suspicious to the least suspicious.

Sample Cases. The technique by Debroy and Wong might be able to fix the faulty programs that depend on the use of incorrect conditions and arithmetic operators, which are Algorithms 3 and 5. PACHIKA and KALI are ineffective with the sample set of faults presented in Section 4. The former technique is ineffective because the program is not an object-oriented program and thus none of the presented faults is in the scope of the technique. The latter technique is ineffective because none of the faults could be fixed by just dropping functionalities. Since AE shares its search space with the RSRepair search-based technique, it can address the same faults, thus it can repair the fault reported in Algorithm 4.

6.1.2 Pre-Defined Templates

Repair techniques based on pre-defined templates modify programs according to a set of change operators that can affect one or more statements of the program (see row *pre-defined* in Table 2). Using templates developers may define complex change patterns that *coherently* affect the program under repair in multiple locations, which would be hard to obtain with a randomized combination of atomic changes. Examples of these templates are changes that expand synchronization blocks, perform non-trivial manipulations on program conditions, and add code implementing pre-defined access control policies.

So far, search-based techniques in this category exploited templates to address specific classes of faults, while brute-force techniques exploited templates to address both general and specific classes of faults.

Most of the techniques working with pre-defined templates use brute-force rather than search-based strategies. This might be because templates could be more expensive to apply than atomic changes, and this cost factor may hinder the speed of the evolution in search-based algorithms. It might thus be preferable to use brute-force strategies for systematically checking if there is a fault in any place where a template could be applied to.

Search-Based Techniques for Concurrency Faults. In some cases, the application of one template might be insufficient, although useful, to fix a problem. In these cases, search-based strategies might still be viable. A class of problems that might require multiple non-trivial changes in multiple points of a program to be fixed are concurrency faults. For example ARC [38], which is a repair technique for concurrency faults, uses manual templates in the context of a search-based algorithm.

ARC [37], [38] generates candidate solutions for concurrency faults by evolving the program under repair using genetic programming and a set of pre-defined templates, which implement a range of non-trivial changes that might be useful in concurrent programs, including synchronizing unprotected shared resources, expanding synchronization regions to include unprotected source code, and interchanging nested lock objects.

The candidate solutions are evaluated by running the available test suite multiple times, each time injecting a different noise using the ConTest tool [112]. Test cases must be executed multiple times because the capability to observe a concurrency fault might depend on the specific interleaving of an execution, and thus multiple executions of a same test may produce different results. The fitness of candidate solutions is defined as the ratio between the number of correct and the number of distinct interleavings that have been observed.

General Brute-Force Techniques. Brute-force techniques define templates that may fix program faults if applied once to a program. The repairing process is addressed by applying every template to every possible location, until fixing the program or timing out. General techniques define templates that might potentially address any type of fault in a program. In this category, we report the AutoFix-E [92], AutoFix-E2 [93], SPR [28], and Prophet [113] techniques, which exploit general pre-defined templates not bounded to specific fault models.

AutoFix-E [92] is similar to PACHIKA but works on software written in Eiffel, a programming language that supports programming by contracts. The presence of the contracts (e.g., methods pre-conditions and post-conditions) lets AutoFix-E produce fixes that are more complex than the ones produced by PACHIKA. In fact, contracts give semantic knowledge about the program, and can be used to narrow down the set of possible fixes, to differentiate between erroneous and correct program states, and to facilitate the identification of a strategy to reach the correct state.

AutoFix-E modifies the program under repair using the templates shown in Algorithms 7, 8, 9, 10; where *snippet* is new code meant to bring the application in a new state that does not violate the available contracts, *oldStmt* is a statement or block of statements already present in the application's source code, and *fail* is a predicate that captures the sets of values that made the program fail when running the tests in the available test suite.

Algorithm 7. schema 1

```
1: snippet
2: oldStmt
```

Algorithm 8. schema 2

```
1: if(fail){
2:     snippet
3: }
4: oldStmt
```

Algorithm 9. schema 3

```
1: if(!fail){
2:     oldStmt
3: }
```

Algorithm 10. schema 4

```
1: if(fail){
2:     snippet
3: } else {
4:     oldStmt
5: }
```

AutoFix-E uses model-based fix locus localization to determine the points where the templates should be applied, and uses a metric that measures how a fix affects the program source code and the program dynamic state to give priority to the fixes causing the smallest changes. This strategy is based on the assumption that the fix can be obtained with a small change of the program.

AutoFix-E2 [93] (also known as *AutoFix* [94]) refines the way the templates are applied in AutoFix-E taking advantage of the information dynamically extracted from the conditions that are evaluated during test execution in addition to the information extracted from the contracts. Moreover, it extends AutoFix-E with the capability to rank the candidate solutions according to the suspiciousness of the statements in the program under repair computed with spectrum-based fault localization techniques.

SPR [28] addresses program repair with a staged process that can quickly skip many wrong fixes and focus on the most promising cases. The repair strategy systematically applies a set of general transformation templates to the program under repair. These templates are parameterized, hence each one represents a class of program transformations. In a second phase, for each transformed program, SPR determines the parameter values that can make the repair successful, if any. Since the templates often require a condition as a parameter, in the last step SPR attempts to synthesize a condition that can produce the values determined in the second phase when evaluated. If the process is successful, the condition is embedded in the program and the repair is finalized.

The parametric templates defined in SPR may produce changes that affect *if* conditions (by adding clauses to existing conditions and by generating new conditions) and variable values (replacing variable names and constant values). The synthesis of new conditions is limited to conditions of the form *v op const*, where *v* is a variable name, *op* is either = or *!=*, and *const* is a constant value. SPR can also copy the statements present in the source code to other locations in the program.

Note that compared to atomic change operators, the templates defined in SPR may address more complex scenarios. For instance, a template may transform entire conditions, while atomic change operators usually affect the individual operators. Of course, the effect of some of the templates might be potentially recreated by techniques using atomic change operators through evolution.

SPR systematically applies these templates from the most suspicious to the least suspicious statement.

For instance, if we consider Algorithm 2 as a sample program to repair with SPR, the repair process would first select a statement and a template. Let us assume the `if` statement at line 1 and the template that adds a new clauses are selected. The new condition would be of the form `a == 0 && b == 0 || absCond`, were `absCond` has to be determined. SPR runs the available test suite and checks what Boolean values returned by the expression `absCond` would turn the failing test cases into passing test cases, without changing the outcome of the passing test cases. SPR also records the values of all the variables in the scope of the faulty `if` statement to synthesize a condition that uses these variables and returns the desired sequence of Boolean values, in this case successfully identifying `a==0` as the right clause to add.

Prophet [113], [114] uses the same fix generation process of SPR, but it improves the repair process exploiting the information available on a large database of software revision changes (containing many fixed programs). The intuition is that the same kinds of fixes may reoccur across the lifetime of different projects, thus it is possible to learn from the past to make the repair process more efficient. In addition, learning from human-written fixes increases the likelihood that the generated fixes are correct and not only plausible.

Prophet analyzes the database of software revision changes to produce a probabilistic model $P(m, l|p)$ that encodes the probability that a repaired program can be derived from a program p with an AST modification m at modification point l. These probabilities are used to rank the candidate patches and try the patches that have higher probability to fix the program first. This strategy compared to the set of heuristics used by SPR may significantly reduce the time necessary to explore the search space.

Sample Cases. AutoFix-E and AutoFix-E2 target Eiffel programs, thus they cannot address the sample programs.

SPR and Prophet can repair many of the sample faults described in Section 4. In particular, they can repair the fault in Algorithm 2 as exemplified when presenting SPR, but they can also repair the Algorithms 3 and 6 using templates for variable values. They can also repair the fault in Algorithm 4 exploiting templates for copying existing statements (`exit(0)` in this case). They cannot fix the fault in Algorithm 5, because the templates for *if* statements are insufficient to fix the faulty condition.

Brute-Force Techniques for Buffer Overflow Faults. PASAN [115] is a technique designed to repair buffer overflow vulnerabilities. It works by first detecting the inputs used in control-hijacking attacks and then using these inputs to generate fixes that remove the vulnerabilities exploited in the attacks. PASAN can handle three types of buffer overflow vulnerabilities: overflows caused by the use of an unsafe *libc* function (such as *strcpy*), overflows caused by an array copying loop that ends up corrupting a return address, and buffer overflows that do not corrupt any return address.

PASAN instruments the application under repair to extract information about the size of static arrays and dynamically allocated buffers and uses RAD [116] to detect buffer overflow attacks that corrupt a function's return address. At each iteration, based on the nature of the statement that likely introduced the corrupted value, PASAN uses different strategies to produce a candidate fix. If the statement that modifies the return address is a *libc* function, PASAN attempts to fix it by replacing the unsafe function with a safe one. If the statement that corrupted the memory address is in a loop, PASAN attempts to fix it by first identifying the array being overflowed and its size, and then enforcing the bound checking on such array with an *if* statement. Finally, if the vulnerability is not on a return address, PASAN attempts to find the library function or the loop that originated the problem and change it by introducing appropriate checks in the code. The generated fix is tested by replaying the attack against the fixed program.

AutoPAG [117] uses an approach similar to PASAN to repair out-of-bound violations. It first instruments the application to detect the variables that overflow and thus identifies the *tainted sets* of statements and variables. The fix generator works on these tainted sets using different fix templates: redirecting an out-of-bound read within the buffer boundary, replacing a call to a function that allows out-of-bound writes (such as `strcpy`) with a call to a safe function (such as `strncpy`), and simply skipping the statement that causes the out-of-bound violation. The fixed application is then tested against the same out-of-bound exploit that triggered the repair process.

6.1.3 Example-Based Templates

Repair techniques using example-based templates modify programs, either evolving them (search-based techniques) or systematically performing changes (brute-force techniques), according to a set of change operators that are extracted from a sample set of fixes that have been already used to fix programs (see row *example-based* in Table 2). Example-based templates, similarly to pre-defined templates, might implement quite complex schema that coherently affect the program under repair in one or more locations.

The extraction of the templates might be manual or automated. When the extraction is manual, the templates are defined once for all and then used to fix programs. This is the case of History-driven repair [118], PAR [22], Relifix [119], and R2Fix [120]. Note that History-driven repair, although it automatically mines information for guiding the evolution of the candidate solutions from historical data, actually uses a simple pre-defined set of atomic operators and exampled-based templates as change operators.

In other cases, the templates might be extracted automatically using mining techniques or other algorithms. In this case, the extraction process can be repeated every time from a different set of programs, increasing the generality of the technique. This is the case of CodePhage [24], which can automatically extract fixes for buffer overflow problems from a set of correct programs. The effectiveness of

example-based techniques is clearly dependent on the set of cases used to extract the templates [121].

General Search-Based Techniques. The techniques belonging to this category do not consider a specific class of faults (their fault model is general) but use a specific change model derived from the analysis of several real-world fixes. The extracted templates are automatically recombined by search-based algorithms in order to maximize their effectiveness in repairing faults, assuming some faults may require the simultaneous application of multiple templates.

History-driven repair [118], [122] exploits information extracted from the history of several software projects to guide the generation of candidate fixes. The technique uses the same evolutionary approach of GenProg, but without a crossover operator and with 12 mutation operators derived from GenProg [20], PAR [22], and mutation testing [123]. The information mined from the projects is used to synthesize bug fix patterns that represent sets of AST-level changes that have been useful to fix bugs in the past. The bug fix patterns are used to guide the selection process, that is, the candidate solutions that are most likely evolved in each iteration are the ones that incorporate a set of changes similar to the ones represented in the bug fix patterns.

PAR [22] uses templates defined from the manual analysis of more than 60,000 real-world fixes. The assumption is that templates defined from real-world fixes might be more effective than other templates and may also improve fix acceptability.

Templates are encoded as sequences of AST rewriting rules and are used by a genetic programming algorithm to evolve a set of candidate solutions. Templates are applied to code locations determined according to spectrum-based fault localization until a fixed program is found.

Relifix [119] applies an approach similar to PAR in the specific setting of regression problems. Relifix uses a set of code transformation templates defined from the manual analysis of 73 real regression problems. The templates encode rules specific to regression faults, such as replacing a statement with the previous version of the same statement, or mutating a statement that has been just modified.

These operators are used with an optimized random search algorithm that exploits spectrum-based fault localization to identify the code locations that should be targeted with the templates.

Sample Cases. History-driven repair can repair the faults in Algorithms 2, 4, and 6. To fix Algorithm 2, History-driven repair can use the mutation operator that removes a Boolean condition from a composite `if` condition to drop the condition `b == 0`. To fix Algorithm 4, History-driven repair can use the mutation operator that inserts statements found elsewhere in the program that must be fixed. In this case, it can repair the fault by copying the `exit(0);` statement in the `then` branch of the `if` condition at line 1. Finally, Algorithm 6 can be fixed with the replace method call parameter operator (assuming the technique would handle C functions similarly to Java methods) that can replace a parameter in a method call with a variable found in the same scope of the method call. In this case a fix can be obtained by replacing parameter `a` with variable `b`.

For both PAR and Relifix, their ability to fix faults is directly related to the matching between the fault that must be repaired and the templates defined in these techniques. For example, PAR does not include templates that may add a method call, which would be useful to fix the fault in Algorithm 4, while PAR includes a template that can add and remove terms from a condition predicate, which can be used to fix the fault in Algorithm 2 by removing the condition `b==0` from the `if` statement at line 1. Algorithm 6 can also be repaired by PAR using a template that replaces a method call parameter with a compatible one in the same scope, while there are no templates for Algorithms 3 and 5.

The ability of Relifix to deal with the sample faults presented in Section 4 strongly depends on whether those faults have been introduced as regression problems or not. For instance, if any of the faulty statements (e.g., the `if` condition at line 1 of Algorithm 2) have been introduced by simply changing a previously correct version of the algorithm, Relifix may selectively revert the change just for the faulty statement.

General Brute-Force Techniques. The techniques belonging to this category are designed to be able to potentially repair any class of faults, as long as a suitable set of real-world fixes to learn from are provided. Differently from other techniques that use templates manually extracted from a set of real cases once for all, general techniques can extract templates automatically every time from a different set of samples.

R2Fix [120] is a repair technique that generates fixes starting from bug reports filed by users, exploiting a range of pre-defined fix templates associated with a number of real-world bug reports. R2Fix uses machine learning techniques to identify the real-world bug reports that look similar to the bug report of the fault that must be fixed. The identified bug reports are analyzed (e.g., looking at pointers and function names), automatically paired with the associated pre-defined templates that are contextualized to the new bug reports (e.g., although similar, the two bug reports may refer to different variables and function names), and applied to the code.

The identified bug reports and the corresponding templates are applied systematically to the code of the program under repair, and the candidate solutions are validated running the test suite attached to the bug report.

Sample Cases. For the techniques in this category, their ability to address faults depends on the set of samples to learn from. In principle, if suitable samples fixes are provided, any fault could be fixed.

Brute-Force Techniques for Buffer Overflow Faults. The techniques belonging to this category are designed to repair buffer overflow vulnerabilities using templates whose definition is at least partially influenced by real-world fixes and real-world programs. These templates are systematically applied with brute-force algorithms.

CodePhage [24] targets buffer overflow problems, but instead of defining a pre-defined set of templates, uses a set of *donor* programs to extract the conditions that should be added to the program under repair to prevent the buffer overflow problem. The set of donor programs must be programs that implement the same functionality of the program under repair. The assumption of this repair technique is that there might exist a donor program that contains the check that is missing in the faulty program and that can

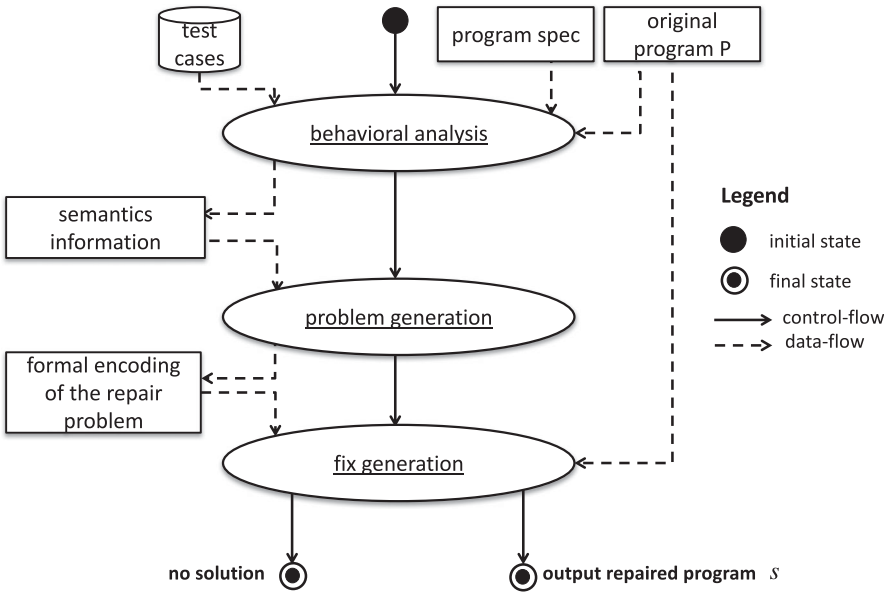

Fig. 7. Semantics-driven repair process.

repair the fault if copied from the donor to the program under repair.

CodePhage uses the (Directed Integer Overflow Discovery Engine) DIODE fix locus localization technique [124] to specifically discover the inputs that trigger the overflow failure in the program under repair. DIODE works in two main logical steps. In the first step, it uses dynamic taint analysis built on top of the Valgrind framework [125] to identify the memory allocation sites in a program and the size of the allocated blocks. In the second step, it uses symbolic execution [126] to analyze the allocation sites and identify the input values that may make the program fail with an Integer overflow. DIODE confirms the presence of the overflow by running the program with the identified inputs and checking the generation of the failure. The target statements for the generation of a fix are the memory allocation sites that have been discovered during the first step and that have been confirmed to cause Integer overflows with the second step.

Note that these statements, although producing the overflow when executed, are not necessarily faulty. Indeed, the values responsible for the overflow could be generated by other statements. However, Integer overflow problems do not necessarily need to be addressed looking at the fault location, but could be prevented directly on the location where the overflow can be observed.

The set of donor programs are then executed with the same inputs to discover a donor program that can process correctly both the error-triggering inputs and the non-error triggering inputs. The conditional branches of the donor program that are executed by the input values are analyzed to discover what are the conditions that evaluate differently on the failing and passing inputs. These conditions are the ones that are likely to significantly affect the failing executions and could be exploited to repair the faulty program.

If such conditions are discovered, CodePhage runs a process to adapt the condition discovered in the donor program to the program under repair. In particular, the condition is first expressed symbolically with respect to the input values of the program, so that it can be transferred into the program under repair. CodePhage then determines every possible place executed by the failing inputs where the condition could be inserted to fix the program, thus generating many program variants that are tested with the available test suite and checked with DIODE.

6.2 Semantics-Driven Repair

Semantics-driven techniques encode the problem P_{repair} formally, either explicitly, for instance as a formula whose solutions correspond to the possible fixes of the program under repair, or implicitly, as an analytical procedure whose outcome is a fix. Thus, a solution s_{repair}, when it can be found, is guaranteed to solve the problem P_{repair} and thus does not need to be validated against P_{repair}.

Note that although a solution s_{repair} is guaranteed to solve P_{repair}, s_{repair} is not guaranteed to be fully satisfactory for the developers. P_{repair} is an approximated representation of the real repair problem to be solved, and a solution might be problematic according to aspects not represented in P_{repair}. For instance, P_{repair} may represent a concurrency problem present in a program and an actual solution to P_{repair} may fix the concurrency problem while introducing other problems, such as performance or functional problems. Thus, the automatically generated solutions still need to be validated manually or automatically to decide if they can be finally accepted.

The general process of a semantics-driven technique is illustrated in Fig. 7. The process consists of three main sequential activities. The *behavioral analysis* activity analyzes the program under repair to extract semantics information about the correct and faulty behaviors of the program. To this end, behavioral analysis may exploit a number of sources, such as the available test cases, a specification, and the source code of the program under analysis. Behavioral analysis typically exploits a subset of these sources. For example, it can focus on the dynamic information that can be extracted by running the test suite or the information that can be obtained by statically analyzing the source code of

TABLE 3
Semantics-Driven Techniques

Fault model	Change model	Tecqhniques	Section
general	synthesis of new expressions	SemFix, DirectFix, Angelix, SearchRepair	Sec. 6.2.2
wrong conditions and missing preconditions	condition change and if condition insertion	NOPOL, Infinitel, DynaMoth	Sec. 6.2.3
concurrency faults	critical region manipulation, parallelization keyword move	AFix, CFix, HFix, Surendran et al. Axis, Grail, Lin et al., DFixer	Sec. 6.2.4
HTML generation faults	string modification	PHPQuickFix and PHPRepair	Sec. 6.2.5
string sanitization	insertion of checks, string modification	SemRep, Yu et al.	Sec. 6.2.6
access control violations	insertion of role checks	FixMeUp	Sec. 6.2.7
memory leaks	insertion of `free()` statements	LeakFix	Sec. 6.2.8

the program. In the former case, the extracted information can be in the form of input-output pairs that encode how a likely faulty code fragment of the program should behave to run correctly. In the latter case, the extracted information can represent the behaviors that should be eliminated or modified from the program under repair to let it run correctly.

The *problem generation* activity exploits the information collected with behavioral analysis to generate either explicitly or implicitly a formal representation of the repair problem whose solutions are code changes that represent actual fixes.

The *fix generation* activity tries to solve the problem generated in the previous step, either identifying the code change that may fix the program, or producing no solution in the case the solution to the repair problem does not exist or cannot be found in a reasonable amount of time.

In some cases, the problem generation and fix generation activities might be iterated considering different program locations as a target for the fix. The fix generation process may also involve implicitly defining and traversing a fix space driven by the specific formulation of the repair problem.

Semantics-driven techniques often address specific classes of faults rather than being general. This is because it is easier to find a formal representation of the problem P_{repair} when a specific characteristic of a program under repair is considered (e.g., the locking discipline), than trying to produce a fully comprehensive formalization of the repair problem. Table 3 shows the classes of faults that can be addressed with existing semantics-driven techniques (column *Fault model*), the code elements that are manipulated in the attempt to repair a fault (column *Change model*), and the corresponding repair techniques (column *Techniques*). Column *Section* indicates the section of the paper that discusses the techniques listed in the corresponding row. In the following, we first provide a short introduction to program synthesis, which is used by several semantics-driven techniques, and then we discuss the techniques in each category.

6.2.1 Program Synthesis

A key enabler of several semantics-driven techniques, especially the ones with the *general* and *wrong conditions and missing preconditions* fault models, is program synthesis, which is exploited to effectively construct a fix for the program under repair. The specific program synthesis approach used by these semantics-driven techniques is the oracle-guided component-based program synthesis [127]. In the following, we briefly present this form of synthesis.

The synthesis process takes as input both a set of input-output pairs that the synthesized program must satisfy and a set of basic components that can be recombined to generate the program (e.g., operators, constant values, and functions), and outputs a function (or a program) that satisfies every pair.

When used as part of a statement in a synthesized program, every basic component defines a variable and uses one or more variables, which can be defined by other basic components (e.g., the + basic component uses two variables, which are the two numbers that must be summed, and produces one variable, which is the sum). The synthesis process generates the required program by identifying a suitable set of connections among the variables and the components. This problem is typically encoded as a first order logic constraint problem and is solved using a Satisfiability Modulo Theory (SMT) solvers.

For instance, if we suppose to provide a synthesis algorithm with two input variables `x` and `y`, with the basic components `C = {+, sqr, sqr}` and with the set of input-output pairs `I = {{(0,0),0}, {(1,0),1}, {(1,1),4}, {(2,2),36}}`, the synthesis algorithm tries to connect the components in a way that satisfies every input-output pair. Fig. 8 shows both an example program that can be created through synthesis and a visual illustration of how the program can be produced by connecting input variables and components. In particular, the components are indicated with rectangles and identified by a

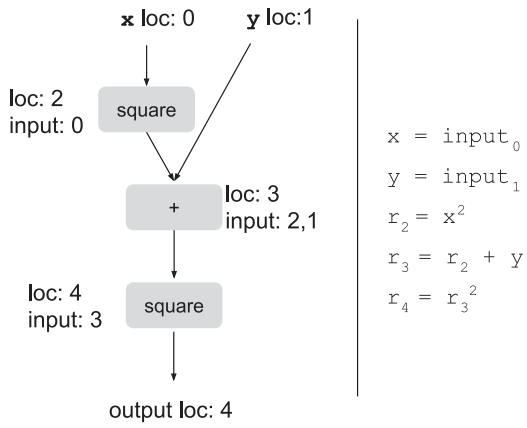

Fig. 8. Example program created with program synthesis.

location number (see label `loc`), edges indicate how the value produced in a given location is used as input in another location (see label `input` associated with every location). The SMT solver is used to find these connections, that is, finding an appropriate value for the variables `input` associated with every location.

Depending on the set of basic components that are used, programs of various complexity can be generated.

6.2.2 General Techniques

SemFix [21] synthesizes fixes that consist of a single changed statement. Thus, fixes that require changes to multiple locations of the code are out of the scope of this technique. To identify the statement that should be changed, SemFix uses spectrum-based fault localization and considers statements in order of suspiciousness, starting from the most suspicious one. Every time a statement is considered, SemFix tries to synthesize a fix by modifying a branch predicate, that is, changing $conditional(f_{faulty}(...))$ into $conditional(f(...))$, or changing the right hand side of an assignment, that is, changing $x = f_{faulty}(...)$ into $x = f(...)$.

In practice, SemFix replaces the expression in the program under repair with a symbolic expression that represents either a generic condition or a generic assignment of a value to a variable. When the available test cases are executed, the program is executed concretely until the modified statement and then symbolically. These executions are used to produce a set of constraints on the symbolic expression that has been introduced in the program; the constraints encode the conditions that the symbolic expression must satisfy to make the program pass all the available test cases.

These constraints together with a set of basic components are used to solve a program synthesis task. In order to keep the generated expression as simple as possible, the basic components supplied to the synthesis algorithm are divided in levels and fed to the synthesis process incrementally. For instance, when synthesizing conditional statements the first basic components that are used are constants, then the algorithm progressively adds comparison operators, logic operators, arithmetic operators, and so on.

If we consider the faulty program shown in Algorithm 5, SemFix would consider changing the statements of the program according to their suspiciousness. Let us assume at some point SemFix targets the statement at line 1. SemFix tries to produce a new statement of the form `if(f(...))` where `f` is a function of the input and program variables. Since only two variables `a` and `b` are defined at that point of the program, the signature of `f` must necessarily be `boolean f(a,b)`. Let us assume that the execution of the available test cases produces the following constraints `f(10,15)= false`, `f(0,10)= true`, `f(0,15)= true` that must be satisfied to pass the test suite. A possible solution to this synthesis problem is `a==0`. Thus the program can be fixed by replacing the `if` condition at line 1 with `if (a==0)`.

DirectFix [128] repairs faulty expressions inside a program with a monolithic approach that is technically similar to SemFix. DirectFix translates a faulty program into a *trace formula* f that encodes its behavior, then it translates the set of failing test cases into a set of *oracle constraints* O that are unsatisfiable if conjuncted with the trace formula. The goal of the repair process is therefore defined as the attempt to modify the expressions in f such that $f \wedge O$ is satisfiable. This is done by reducing the problem into an instance of the partial MaxSAT problem: the basic expressions in f are extracted, conjuncted with the original formula, and replaced with placeholders generating what is called *repair condition*; a partial MaxSMT solver is then used to generate a new formula f' which satisfies $f' \wedge O$. If a solution is found, the changes at the level of the formula are retrofitted to the code.

When multiple fixes are possible for a same faulty expression, DirectFix selects the simplest fix, under the hypothesis that simple fixes are less likely to introduce regression problems.

Compared to SemFix, DirectFix aims at reducing the complexity of the fixed expressions, and improving fix acceptability.

Angelix [95] is a semantics-driven repair technique that aims at synthesizing multi-line fixes while preserving scalability, in contrast with DirectFix, which can produce multi-line fixes but does not scale well, and SemFix, which is scalable but limited to single-line fixes. To generate multi-line fixes without sacrificing scalability, Angelix exploits the concepts of *angelic path* and *angelic forest*. An angelic path encodes part of the repair problem as a set of triples each one containing an instance of a suspicious expression (identified with spectrum-based fault localization), its angelic value (the value that the expression should return to pass the tests) and its angelic state (a set of variables visible at the location of the expression). Angelix paths are extracted using symbolic execution. An angelic forest fully encodes the repair problem as a set of angelic paths. The angelix forest is fed to a fix synthesis engine to produce multi-line fixes.

SearchRepair [44] exploits a database of human-written patches encoded as SMT formulas. This results in a database of SMT formulas encoding changes that have fixed faults in various cases. When a program must be repaired, SearchRepair uses spectrum-based fault localization to determine the likely faulty code fragments, while the failing and passing test cases are used to generate an input-output constraint that describes the desired behavior for each possibly faulty code fragment. The database of SMT formulas is then searched looking for a change that might produce the desired input-output behavior for the target fragment. If it is found, the change represented in the database is performed in the program under repair. This process is repeated systematically for every region that has been selected. The candidate fixes obtained in this way are validated with the available test suite.

Sample Cases. SemFix, Angelix, and DirectFix can repair the faulty programs in Algorithms 2, 3, and 5. The ability of SearchRepair to fix the faulty example algorithms depends on the sample fixes that the database contains: if suitable code fragments are provided, any fault could be fixed.

6.2.3 Wrong Conditions and Missing Preconditions

The techniques presented in this section target faulty conditions (occurring either in a branch or in a loop statement) and missing preconditions. Although each technique uses specific mechanisms, the shared high level process consists of identifying the most suspicious conditions in the program

by using localization algorithms, gathering variables that are available in the scope of the suspicious condition, and formulating a program synthesis problem, whose solution is the repaired condition.

NOPOL [23], [129] synthesizes fixes that consist of a single changed condition occurring in a conditional or loop statement. NOPOL uses spectrum-based fault localization to identify the suspicious statements that include conditions and angelic fix localization to confirm the existence of a potential fix location (see Section 5). In practice, angelic fix localization can confirm that negating the truth value of the selected condition for the failing test cases can repair the program.

NOPOL runs the test cases and collects evidence about the truth value that the target condition must return to pass all the test cases, together with the values of the variables in the scope of the likely faulty condition that must be repaired. The truth values that must be returned by the target condition can be represented as a set $O_{l,m,n}$ whose values represent the expected outcome of the condition at program location l the mth time it is evaluated during the execution of the nth test case. The values of the variables in the scope of l are represented by the $C_{l,m,n}$ set whose values represent the collected variable values at location l during the mth execution of the nth test case. The repair synthesis problem can thus be encoded as finding an expression exp that satisfies

$$\forall(l, m, n)\ exp(C_{l,m,n}) = O_{l,m,n}$$

Possible values for exp are obtained by combining simple arithmetic expressions and program variables.

If applied to the faulty program shown in Algorithm 5, angelic fix localization can easily identify the statement at line 1 as the statement to be modified to make the program pass all the available test cases. The execution of the test cases produces a set of observations $C_{1,1,1} = (a = 10, b = 15)$, $C_{1,1,2} = (a = 0, b = 10) \ldots C_{1,1,n} = (a = 0, b = 15)$ and the corresponding expected outcomes $(O_{1,1,1} = false, O_{1,1,2} = true, \ldots, O_{1,1,n} = true)$ which are used as input to the program synthesis procedure. A solution to this synthesis problem is the expression `a==0`, which is used to fix the program by replacing the condition used by statement 5 of the program.

Dynamoth [130] is a new synthesis engine for NOPOL. Since NOPOL cannot synthesize conditions that include method calls, Dynamoth extends NOPOL with the ability to collect the runtime context of the suspicious condition, including parameters, variables, fields, and return values of method calls, and to combine these elements in a newly synthesized condition.

Infinitel [26] is similar to NOPOL but specifically targets infinite loop faults, thus its repairing process is tailored to change loop conditions to run the loop a finite number of times while making the program to produce the expect output.

Infinitel assumes a loop to be faulty if it is executed more than one millions of times. Exploiting the same strategy implemented in angelic fix localization, Infinitel determines the number of times that the loop should be executed to make the test case pass, if any. This number is called *angelic record*. Similarly to NOPOL, Infinitel generates a set of constraints forcing the new condition of the loop to evaluate to true while the number of iterations is smaller than the angelic record and evaluate to false when the angelic record is reached. Given this set of constraints, the synthesis of the new condition works the same than in NOPOL.

Let us consider the faulty program shown in Algorithm 3. When executed with the test input (`a` equals to 10 and `b` equals to `0`), the program loops indefinitely on the loop statement at line 4. Infinitel analyzes the execution and discovers that the failing test case can pass for an angelic record equals to `0`, that is, if the loop is never executed. This evidence, together with the data collected from the other test cases, is used as input to a program synthesis procedure whose solution is the right condition for the program.

Sample Cases. NOPOL and Dynamoth target faults due to faulty conditions, thus they can only repair the faulty programs in Algorithms 2 and 5, while Infinitel targets infinite loop faults, thus it can repair the faulty code in Algorithm 3.

6.2.4 Concurrency Faults

The techniques presented in this section target concurrency faults, such as atomicity violation, deadlock, livelock, and safety policy violation. The general approach of the techniques addressing concurrency faults consist of analyzing the system to discover the weak code regions, that is, those code regions that might be responsible for concurrency faults, and then strengthen these code regions by adding proper checks and synchronization mechanisms that may prevent concurrency problems.

AFix [82] can repair single variable atomicity violations faults, that is, faults due to operations that are supposed to be atomic with respect to a variable, but turn out they might be interleaved with other operations accessing the same variable.

As part of behavioral analysis, AFix uses CTrigger [83] to monitor normal program executions and discover atomicity violations, which are represented as triples (p, c, r), where p and c are instructions sequentially performing operations on a variable x on a same thread, and r is a remote modification/read of x that happens between p and c.

In these cases, AFix generates a fix by solving the simple problem of putting p and c in a critical region and r in another critical region controlled by a same lock to make the two regions mutually exclusive. This process is applied multiple times for all the atomicity violations faults that have been discovered. AFix includes a final step to remove redundant fixes and merge overlapping fixes.

CFix [131] has been designed to have more general repair capabilities than AFix, which can only address atomicity violations by working with mutex lock primitives. CFix addresses concurrency bugs by decomposing them into a combination of mutual exclusion and ordering problems. The mutual exclusion problem is the same considered in AFix, while the ordering problem is introduced in CFix.

Starting from a bug report, CFix first uses a variety of bug detectors for concurrency bugs to find the failure-inducing interleavings and maps a discovered bug to a specific mutual exclusion or ordering problem. CFix then uses static analysis to determine where to use locks and condition variables to synchronize program actions and potentially repair the bug. Depending on the type of problem, CFix uses either AFix to enforce mutual exclusion or a new technique, called

OFix, to enforce order relationships. CFix may identify a few fixes for a same concurrency problem due to the availability of multiple repair strategies for some classes of bugs. These fixes are tested for performance and simplicity and the best one is selected. Finally, the fixes produced for multiple bugs are merged to introduce the smallest number of synchronization variables and operations in the program under repair.

HFix [132] is an extension of CFix that addresses the problem of fixing concurrency bugs by only adding lock operations and condition variables unnecessarily increasing the fix complexity in some cases. Both these approaches are designed to solve an interleavings problem identified from the analysis of the program. HFix produces simpler fixes by largely reusing the code that is already present in the program rather than generating new code. The technique consists of two bug fixing approaches: $HFix_{join}$ and $HFix_{move}$. $HFix_{join}$ can fix an order violation fault that involves two threads by inserting joining operations that enforce the correct order of the operations. $HFix_{move}$ can fix both order violation faults by re-arranging the ordering of the operations with respect to join operations moving operations to a new thread if necessary, and atomicity violation faults by moving lock and unlock operations to establish appropriate critical regions.

Surendran et al. [133] present a technique that aims to repair data races in programs written with structured parallel languages, specifically targeting the usage of two constructs that these languages use to achieve parallelization: *async* and *finish* (used for creating and terminating tasks, respectively). The technique takes a program that is not or is partially synchronized and a test case as input, uses the ESP-bags algorithm [134] to detect data races, and identifies where to insert additional *finish* statements that can prevent the generation of the discovered data races using a mix of dynamic and static analyses that contribute in the formulation of the repair problem.

Axis [135] addresses atomicity violations but assumes that the statements involved in atomicity violation have been already identified in some other way, that is, it does not implement the behavioral analysis activity. Axis creates a Petri Net model of the program, and encodes the correct behavior of the software as control constraints that can be solved mathematically to obtain a new Petri Net model where the atomicity violation has been removed without introducing any deadlock.

The new locks introduced in the generated Petri Net model are added to the source code of the program to fix the concurrency fault.

Grail [136] is a technique that extends Axis with the ability to generate fixes that are not only correct but also optimal in the sense that mutual exclusion is applied only in the exact scenarios when the bug may manifest. The fixes produced by Grail are also more efficient than the fixes produced by Axis because Grail takes into account both the multiple runtime configurations that can expose concurrency failures and the synchronization behavior of the threads. The analysis is still based on Petri nets models.

Lin et al. [137] present a technique that can repair deadlocks, livelocks, and a third kind of starvation faults called deadlivelock, which is a subtle blocking situation with both threads blocked waiting for locks and threads actively trying to acquire locks. The approach used by this repair technique is to first identify the sets of cyclic dependent statements and all the *lock* and *trylock* statements in the program. Then it encodes the desired program behavior as a weighted partial MaxSAT formula that is solved looking for solutions that minimize the size of the fix. If a solution is found, the original program is modified accordingly, for instance substituting lock with trylock and removing the *false* branch of a trylock operation.

DFixer [53] can repair deadlocks without any risk to introduce new deadlocks. The technique first analyzes the program to identify where pairs of threads can create a deadlock by acquiring resources in reverse order. This information is exploited to construct a representation of the program behavior and of the repair problem to be solved. When a problematic situation is identified, DFixer selects one of the two threads involved in the deadlock and replaces the first lock operation that the thread performs with one that also acquires the second lock. For example, if $thread_1$ tries to acquire in succession two locks on two different resources r and s with operations $acquire(r)$ and $acquire(s)$, and $thread_2$ tries to acquire the same resources but in reverse order, a possible fix is to replace $acquire(r)$ and $acquire(s)$ in $thread_1$ with $acquire(r, s)$.

6.2.5 HTML Generation Faults

HTML Generation Faults are faults in html generating code. *PHPQuickFix* is a simple technique that can detect and repair print statements that print literals with incorrect html code [138].

More interesting *PHPRepair* can address faults caused by the interaction of multiple constant print statements, such as the presence of a missing $<$ `/td` $>$ tag in a dynamically constructed HTML table [138]. PHPRepair first runs the available test cases and characterizes test executions as sequences of constant prints. These print statements are then compared to the expected output according to the test cases. This produces an encoding of the repair problem that is solved by a string constraint solver that identifies what the correct stream of prints should be. Finally, PHPRepair adds, modifies, and deletes constant print statements in the original program to match the expected behavior.

6.2.6 String Sanitization

String sanitization techniques repair routines that incorrectly check the validity of input strings in Web applications.

SemRep [139] is a technique that can repair partial input validation functions present in Web applications by automatically adding the checks that are present in other validation functions but missing in the target function. The idea is that similar validation functions might be present within the same application and in different applications and they can be exploited to repair incomplete validation functions by transferring checks from one function to another. The technique uses forward and backward symbolic string analysis to formulate the repair problem and transfer the validation, length, and sanitization checks.

Yu et. al [52], [140] present a technique that generates fixes for input sanitization functions in Web applications. The technique starts from an input pattern and an attack

TABLE 4
Fix Recommender Techniques

Fault model	Techniques	Section
general	BugFix, MintHint, Logozzo et al., QACrashFix	Section 7.1
security faults	BovInspector, Abadi et al., CDRep	Section 7.2
data type misuses	Coker et al., Malik et al.	Section 7.3
concurrency faults	ConcBugAssist	Section 7.4
performance faults	Selakovic et al., CARAMEL	Section 7.5

pattern (specified as a regular expression) that breaks the application and generates a fix in two phases. The first phase, *sanitization signature generation*, automatically identifies the signature of benign input strings. It first uses symbolic string analysis to compute an over-approximation of every possible string that can reach security sensitive functions, that once intersected with the attack pattern indicates the set of the possible attack strings. Backward symbolic reachability analysis is then used to identify the malicious user inputs that can produce these attack strings. The set of safe inputs, that is, the sanitization signature, is given by the inputs that are not in the set of the malicious user inputs and that can be constructed with the application.

The second phase, *optimal sanitization synthesis*, generates a sanitizer that automatically converts the malicious input strings into benign strings while minimizing the degree of intervention on the input according to an edit distance.

6.2.7 Access Control Violation

FixMeUp [42] specifically targets access control violation faults in PHP Web applications. This technique works by detecting the vulnerable program locations, and then transforming the program sections that lack proper access control using an access control template.

FixMeUp requires a specification that indicates which sensitive operations must be guarded by access control checks, then it uses inter-procedural program slicing on the call, data, and control dependence graphs of the program to identify the statements that need to be guarded by an access control check.

FixMeUp checks that for every sensitive operation and for every calling context, only the user roles that are allowed to execute the sensitive information can indeed do that. If a violation is found, an appropriate template is used to fix the program by inserting the missing access control check.

Finally, to detect the side-effects that might be introduced by the repair, the analysis performed to reveal the violations to the access control rules is repeated to check if the problem has disappeared. If the problem has not been fixed, the change is discarded. This process is repeated for every problematic point in the program.

6.2.8 Memory Leaks

LeakFix [54] targets memory leaks in C programs. It works by analyzing memory allocation statements and making sure that there is no leak in any execution path. This is done by first identifying the functions that allocate, use, or deallocate memory, and then abstracting the program with a control flow graph that contains information about allocations, uses, and deallocations. If a leak is detected, a *free()* statement is added to a suitable location of the program to deallocate the unused memory.

7 Fix Recommenders

Fix recommenders are techniques that do not attempt to produce fixes, but simply suggest a few changes that might be operated on the software to repair the fault. In some cases the recommended changes might fully describe the required fix, in some other cases some effort might be required to the developers to produce the final fix. Although these techniques do not produce an actual repair, that is, they do not produce a new version of the software that is supposed to be correct, their output can be quickly turned into a fix in the best cases.

A few fix recommender techniques have been proposed so far. Here we discuss these techniques classified according to their fault model. *General* techniques aim to be effective with a range of programs, not limiting their scope to a specific class of faults. The other techniques can address *security* faults, which make programs vulnerable to attacks; *data type* faults, which may cause failures due to the misuse of data structures and other types of data; *concurrency* faults, which may cause deadlocks and other concurrency issues; and *performance* faults, which may cause unacceptable execution time for some functionalities.

Table 4 summarizes the discussed techniques (column Techniques), the classes of faults addressed by each technique (column Fault model), and the section in which the technique is described (column Section).

7.1 General Techniques

BugFix [39] can analyze the debugging information at a specific program statement and report to developers a list of possible actions that may fix the faulty code. The technique exploits spectrum-based fault localization to identify the suspicious statements, and uses static and dynamic metrics to classify and relate the problem that must be fixed to the entries in a database of debugging rules. The fix suggestions extracted from the database are finally reported as a prioritized list to the developer.

MintHint [141] also generates a ranked list of actions that can be performed to fix a fault. MintHint identifies the statements that are likely to be faulty using spectrum-based fault localization. Each faulty statement is replaced by a symbolic state transformer (i.e., a sort of abstract statement defined on the state of the program only) that satisfies the property of making the program pass all the available test cases. MintHint then explores a space of possible changes that can be performed on the faulty statement to obtain the same behavior defined by the state transformer. The possible solutions are finally returned to the developer ranked according to their likelihood of occurring in the repaired statement.

Automatic program repair techniques can be experienced also as fix recommenders if properly integrated with the development environment. This possibility has been

investigated by Pei et al. [142] who integrated AutoFix-E into the EiffelStudio Development Environment to automatically find faults and recommend source code level fixes.

Logozzo et al. [143] present a technique that recommends code fixes at design time for programs enriched with contracts (preconditions, postconditions, and invariants). The generation of a fix recommendation is triggered by verification techniques for checking contracts, *cccheck* [144] in this case. When an assertion does not hold, the technique exploits a range of analyses based on either backward analysis or data type analysis to produce the possible fixes.

QACrashFix [55] automates the fixing process for the faults that cause program crashes by extracting the possible fixes from the code snippets posted by users in Q&A pages. QACrashFix first analyzes the crash report of the application to build a query that is submitted to a search engine to retrieve a set of Q&A pages that are likely related to the fault. The code snippets present in the retrieved pages are then exploited to identify a set of changes that should be implemented in the code to fix the fault. This analysis is guided by natural language keywords that are searched in the retrieved pages, such as ''instead of'' or ''change x to y''. The changes that may fix the program are encoded in AST level edit scripts that can transform the source code of a program. The fix is finally applied by localizing the likely faulty areas of the code that match the faulty code snippets present in the retrieved Q&A pages and by running the AST edit scripts. Note that the transformed code is not validated automatically but it is simply presented to the developers for manual inspection.

7.2 Security Faults

BovInspector [145] uses static analysis and symbolic execution to analyze buffer overflow threats and suggests fixes by applying the following three change strategies: add boundary checks, replace API calls with calls to safer API functions, and modify buffer instantiation. The specific change to be applied to each API call is based on a set of patterns.

Abadi et al. [146] present a technique that recommends fixes for input sanitization routines in Java. The technique uses a commercial tool to reveal vulnerabilities related to input values and recommends fixes for escaping unsafe special characters or, in case of SQL queries, for replacing the unsafe `Statement` type with the safe `PreparedStatement` type. The analysis can consider the existence of trusted values that must not be sanitized.

CDRep [147] targets cryptography misuses in Android applications. In an initial phase it detects misuses in the decompiled Java code using CRYPTOLINT [148]. It then suggests fixes using a pre-defined set of transformation rules that can modify the program by eliminating the detected misuses. CDRep supports seven common cryptography misuses.

7.3 Data Type Misuses

Coker et al. [149] present a technique that can recommend fixes for Integer type misuses in C programs. The technique provides three program transformations that can be executed from the Eclipse IDE: adding explicit Integer cast operations, replacing arithmetic operators with calls to safe functions that detect overflows and underflows, and changing the type of the Integer variables.

Malik et al. [150] propose to exploit the reports generated by data structure repair tools [151] to recommend fixes. The presented technique takes as input a Java method, the structural invariants that the method must preserve when manipulating the data structure, and an input that violates these invariants, and produces a fix recommendation that should prevent the violation of the invariants. The fix is determined by first identifying the correct data structure that should be produced instead of the corrupted one, and then identifying the operations that should be performed to produce such a data structure. Fixes are recommended to users based on a set of predefined cases supported by the technique.

7.4 Concurrency Faults

ConcBugAssist [152] is a technique that diagnoses and suggests fixes for concurrency faults. It applies bounded model-checking to compute the program inputs and to identify the erroneous thread schedules that make the program fail (e.g., the violation of an assertion). It then uses a MAX-SAT solver to compute the minimum set of thread interleavings that causes the concurrency problem. The model checking is then repeated using a *blocking clause* to exclude the identified thread interleaving from the analysis. This process is repeated iteratively until no new failing executions can be generated. At this point, exploiting the blocking clauses that have been generated, the repair problem can be instantiated as a binate covering problem, and the possible solutions, in terms of the order relation enforcing, are reported to the user.

7.5 Performance Faults

Selakovic et al. [153] present a technique that suggests fixes for JavaScript performance faults. The technique is based on 23 recurring fix patterns defined at the level of the AST of the program. Once a pattern has been applied, the original and the changed versions of the program are executed and compared. If the performance of the modified program improves in a statistically significant way, the transformation is recommended to the developer.

CARAMEL [56] is a technique that suggests fixes for a specific family of performance faults: applications that waste time performing the computations inside a loop after the loop condition has become true. These faults are typically fixed by the developers introducing break statements.

CARAMEL addresses these issues by first identifying the statements inside the loop that, under certain conditions, have no effect or useless effect (i.e., they affect variables that are not used later in the computation) outside the loop. It then checks if all the identified statements can be skipped simultaneously, if so, the conjunction of the conditions associated with the individual statements is used as a condition to break the loop. Finally, CARAMEL checks if the loop already terminates when such a condition is satisfied. If it is not the case, CARAMEL reports a performance fault to the user together with a potential fix of the form of a guarded break condition to be included in the loop.

8 Empirical Evidence

Software repair techniques have been evaluated in many diverse contexts, including papers presenting new

approaches and independent empirical evaluations. We analyzed the findings reported in these papers and reconstructed the key empirical evidence that has been collected so far. This evidence is important to understand the status of the research in the area, to plan for additional studies, and to identify future research directions.

We first discuss the benchmarks and the applications and then we discuss the results, distinguishing between the results obtained with generate-and-validate approaches and with semantics-driven approaches.

8.1 Tools, Benchmarks, and Experimental Subjects

A number of automatic program repair techniques have been defined so far, as surveyed in this paper. To facilitate reproduction and comparison among techniques, the availability of tools as well as the existence of benchmarks and shared subject applications are of critical importance. We investigated tool availability by checking if the existence of a publicly accessible tool was reported in the papers presenting an automatic program repair technique. We also checked the Web site of the authors of the papers presenting new techniques and we searched the Web entering tool names in Google. Using this procedure, we have been able to find tools for 46 percent of the techniques surveyed in this paper. We reported the tool name, its website, and the language addressed by the tool in Appendix A. Although several tools are available, it is also true that less than half of the techniques have a tool available on the Web according to our investigation. This may make reproduction of the results and comparison among techniques quite hard in practice.

In terms of the empirical studies, a variety of applications and faults have been used to evaluate automatic program repair techniques. In most of the cases the evaluations involve real bugs in actual systems [154], [155], although seeded faults have been also exploited sometime [156]. When a test suite is necessary for the purpose of the evaluation, test suites written by the developers of the software [154] and test suites automatically generated with testing tools [24], [93], [138] are regularly used, and only the earliest papers in the area considered smaller programs and manually defined test suites for the purpose of the evaluation [35], [64], [97].

Studies focused on the use of unit test cases, with system test cases sometime used for command line applications, while interactive applications equipped with GUI test cases and acceptance test cases have been seldom considered. PHPRepair [138] and R2Fix [120] are two exceptions since they use GUI test cases and test cases similar to acceptance test cases derived from user-supplied bug reports, respectively.

In terms of size, automatic program repair techniques have been already successful with programs with hundreds of thousands of lines of codes, and in a few cases scaled up to pretty large programs with millions of lines of code [154]. This is a good initial evidence of the scalability of the approach, although the range of large applications that have been experimented with automatic program repair techniques is still too limited to conclude anything about their scalability.

In our analysis, we indeed noticed that the range of cases that have been used for the evaluation have been strongly influenced by the publicly available benchmarks. In fact, the set of applications that have been initially used to evaluate GenProg [20] together with the ManyBugs and IntroClass benchmarks [154] have been used as experimental subjects in 32 percent of the papers that we analyzed. If we also include the SIR benchmark [156], this percentage increases to 45 percent. If we exclude the papers just using examples as a form of validation and the papers that address concurrency and security faults, which require specific classes of faults for the evaluation, these two percentages increase to 44 and 62 percent, respectively. We can conclude that, although having benchmarks is important because it facilitates the comparison among techniques, there is also a tangible risk of producing a research that may overfit the available benchmarks. It is thus important to better exploit the set of benchmarks that are currently available (e.g., the CoREBench [155], Defect4J [157], and IBugs [158] benchmarks have been used only in 3, 2, and 2 percent of the papers, respectively) and to make the effort to design and release additional benchmarks that can help achieving general results.

8.2 Evidence for Generate-and-Validate Approaches

We organize the main pieces of evidence that we extracted from the papers on software repair based on generate-and-validate in three sets: empirical evidence about the impact of the test suite, empirical evidence about the impact of the search space, and other empirical evidence on the effectiveness of generate-and-validate approaches.

8.2.1 Impact of the Test Suites

In general, generate-and-validate techniques run test suites to check the quality of the candidate solutions and determine if any plausible fix can be reported to the user. Here we report the main findings about how test suites can influence the repair process.

The size of the test suites may strongly influence the quality of the generated fixes: The behavior of generate-and-validate techniques extensively depends on the available test cases. Having a small test suite that partially covers the behavior of the program under repair may induce a generate-and-validate technique to repair a fault by dropping the functionalities that are not checked by the tests but interfere with the repairing problem [110], [159]. A tendency to produce programs that pass all the test cases by dropping statements has been also reported in [160], where biasing the choice of the mutation operator in favor of the delete statement operator improved the success rate and decreased the repair time of GenProg. On the other hand, having large test suites that extensively (and unnecessarily) cover the execution space of the program under repair may slow down the repairing process [29], [41], [161]. To successfully repair a program is thus important to have test suites that efficiently sample the execution space, covering most of the behaviors possibly without redundancies [20], [22], [27], [29].

The nature of the test suite may impact on the generated fixes: Some studies reported that using test suites with high coverage might be beneficial for generate-and-validate techniques [162]. This may let people believe that adding many

positive test cases to the test suites is a good strategy to build test suites that effectively guide the repair process. This does not seem to be necessarily true. In fact, empirical results have shown that test suites with many positive test cases make the repair process harder [110], [161], only partially compensating this trend with a better quality of the fault localization [161]. On the contrary, there is small evidence that a good number of failing test cases might be sometime beneficial [161].

Preliminary studies investigated the impact of the kind of coverage achieved by a test suite and the effectiveness of the repair process reporting that test suites satisfying branch coverage can more effectively prevent the generation of wrong fixes compared to random test suites and test suites satisfying statement coverage [36].

Running the test cases is the dominant cost of the repair process: When non-trivial programs are analyzed, the main computational cost for techniques evolving a set of candidate solutions is in the computation of the quality of each solution, which implies running all the available test cases for all the candidate solutions [29]. For instance, it has been reported that this cost may amount to the 64 percent of the total repair time for GenProg [41].

This empirical evidence explains why techniques that do not evolve the candidate solutions (e.g., RSRepair and AE), although they are not able to produce fixes resulting from the application of multiple change operators, might still be successful, sometimes being more efficient than algorithms that evolve the candidate solutions (e.g., GenProg). In fact, these techniques only need to run the test cases to check if a candidate solution is satisfactory and tests execution might be interrupted when the first test case fails, thus incurring in significantly lower test execution cost compared to techniques that always execute the whole test suite. The efficient use of the test cases leads to a higher number of program fixes generated and checked.

Fast et al. [163] investigated how to increase the efficiency of the repair process by reducing the number of test cases that must be executed to evaluate the candidate solutions. In particular, they studied the effectiveness of a repair process that evaluates each candidate solution on a random subset of the test cases instead of using the whole test suite. The full evaluation is conducted only if the candidate solution passes all tests in the selected subset. This optimization has been reported to reduce the running time of GenProg by 81 percent. Although this approach introduces noise in the evaluation of the fitness function, the results suggest that this may not be a problem in practice.

How tests cover faults is more important than the complexity of the fault: The success of generate-and-validate techniques have been reported to be strongly influenced by how well test cases sample the faulty behavior, rather than by the complexity of the fault itself [20], [22], [27], [29]. Of course, a fault could be fixed only if the fix exists in the search space of a repair technique, which implies that the fault can be fixed by applying one or a combination of the change operators used by the technique. However, sampling well the fault is a key factor to produce successful repairs.

8.2.2 Search Space

The shape of the search space explored by a repair technique is of crucial importance for its success. Aspects like the density of plausible and correct solutions can be decisive factors influencing the effectiveness of a repair process.

The size and shape of the search space mostly depend on the strategy (e.g., search-based vs brute-force) and the change operators (e.g., atomic vs templates) used to produce the candidate solutions. In this section, we report the main findings about the relation between the search space and generate-and-validate techniques.

Unclear impact of the strategy: The impact of the strategy, either search-based or brute-force, and also the effect of the individual variants of a strategy, such as using genetic programming or random search, are not clear. There is contrasting evidence, sometime showing that one can be beneficial on the other, and vice versa. For instance, random has been reported to be more effective and more efficient in the generation of plausible fixes than genetic programming [29], [164] and vice versa [161].

This partially contradicting evidence might be due to factors that are difficult to control. For instance, the effectiveness of a strategy may depend on the available test cases, the faults and the type of the program under repair. Moreover, techniques usually do not differ only on the search strategy but also implement other heuristics and optimizations that may impact on the observed results. So when different results are reported, it is always hard to isolate the effect of the search strategy from other confounding factors.

Overall, there is still little solid evidence about the search strategy that should be preferred to approach automatic software repair.

Brute-force might be preferable for repairing small programs: Kong et al. [161] reported preliminary evidence that brute-force techniques may benefit from the reduced execution time that often characterizes small programs, because they can efficiently explore a large portion of the search space, compared to techniques that evolve the candidate solutions, which still require executing the whole test suites many times at each iteration and that are frequently unable to compensate this additional cost with a higher effectiveness. While with larger programs, where the exploration performed by brute-force techniques is limited, the difference between brute-force and other approaches is not significant.

The cost of repeatedly running test cases can be at least partially compensated with fitness sampling techniques, which only require executing random subsets of a test suite, as investigated in [87], [163].

This evidence might be an interesting starting point of additional investigations, which are necessary to confirm this preliminary result.

Few correct fixes in the search spaces: While generate-and-validate techniques have been reported to be able to generate plausible fixes, that is, fixes that pass all the available test cases, a careful analysis of these fixes revealed that most of them would be hardly acceptable by the developers [110], [159]. These plausible but hardly acceptable fixes have been obtained due to the weaknesses of the available test cases, which easily allow for program changes that would not be allowed by developers (e.g., dropping functionalities).

A careful analysis of the search space produced by generate-and-validate techniques revealed the presence of few correct fixes, often hindered by the presence of several plausible fixes that could be erroneously returned to the developers before a correct fix can be reached [159]. This evidence raises issues about the properties that the test suites that should be used to guide generate-and-validate approaches should satisfy (see also Section 8.2.1).

A technique with a narrow change model might be preferable when the fix is in the scope of the technique: Generate-and-validate techniques which reuse statements in the same application [20], [29], [111] might have hard time distinguishing plausible and correct fixes, while techniques with a narrower change model [27], [36], [45], [50], [110], although limited to the modifications foreseen when the technique has been designed, have reported good results in term of their capability to produce correct fixes. Developers should likely experience these techniques first, before trying with techniques using a general change model.

Larger search spaces do not necessarily correspond to a more effective repair process: Increasing the search space in principle increases the range of faults that can be repaired with a generate-and-validate technique, but it might be deleterious in practice. In fact, increasing the search space might actually decrease the density of the correct fixes making the whole process less effective, as reported in [159].

Martinez et al. [165], [166] studied the problem of effectively navigating large fix spaces. They extracted the repair actions that have been performed to fix the software in several software versions and defined a probability distribution that identifies which repair actions are more successful at fixing faults. In their study, they observed that the likely-correct fixes are concentrated in specific regions of the search space and that exploiting probabilities to select repair actions might be beneficial in directing the search towards these regions.

Changes can often be assembled from the code that is already in the program: An analysis of the changes in several large Java projects [100] empirically validated the *plastic surgery hypothesis*, which states that the content of new code in an application can be mostly derived from code that already exists in it. This hypothesis is particularly important for general search-based techniques that use change operators that borrow code from different parts of the program under repair. Results show that 43 percent of the application's new code (including fixes) can be reconstituted from existing code in the version being changed. Moreover they found that 30 percent of the new code in commits can be found within the same file.

A similar analysis is presented in [167], where the *temporal redundancy* assumption is validated. A temporal redundant commit is composed of code elements that can be found in previous commits. Results show that 52 percent of commits are temporally redundant.

Overall these results support the idea that general program repair solutions can be defined by reusing the code that is already present in the same program.

Taking statements from other programs may help fixing more faults: Generate-and-validate techniques that reuse statements in the same application when modifying the source code of a program (e.g., GenProg) are limited by the fact that if the application requires a line of code that is not present in its source code, the technique will not be able to find a correct fix. Sumi et al. [168] investigated the actual possibility to generate fixes from statements taken from other programs by abstracting the structure of the borrowed statement and using variable names that appear close to the fault location instead of the variable names originally found in the statement. In their experiments, they observed a 20 percent increase in the number of plausible fixes generated using this approach.

To efficiently select the statements that can be used to fix faults, Yokoyama et al. [169] investigated the idea of using the statements present in the code regions that are similar to the region that must be fixed. An analysis of several bug reports showed that more than 75 percent of the faults can be fixed using code already present in the application in code regions similar to the faulty one, suggesting that this heuristic is potentially useful for the program repair process.

Anti-patterns can be used to reduce the search space by removing the regions that are likely to produce wrong fixes: Tan et al. [170] identified several anti-patterns that define regions of the search space that frequently lead to fixes that are discarded by the developers and used these anti-patterns to prune the search space. Experiments with SPR and GenProg show that anti-patterns have been useful to produce meaningful fixes that are less likely to remove functionalities and are easier to inspect manually.

8.2.3 Effectiveness

Empirical evidence shows that there are other factors influencing the effectiveness of generate-and-validate techniques than the test suite and the search space. This section discusses these factors.

Using templates extracted from human-written fixes improves acceptability: Templates are often defined from fixes implemented by developers. Since these fixes are likely to resemble the way developers change their code, the templates obtained from these fixes are more likely to produce changes that might be immediately accepted by the developers. This aspect has been studied in [22] by involving both computer science students and software developers. Results suggest that templates can produce fixes that are more acceptable than the ones obtained by techniques using atomic change operators.

Spectrum-based fault localization effectively biases the repair process: Most of the generate-and-validate techniques exploit spectrum-based fault localization to determine where a program should be changed at each iteration (see Section 5.1). Biasing the selection of the statements to be modified with spectrum-based fault localization has been consistently reported as useful, compared to an unbiased selection of the point that should be changed in a program [20], [22], [27], [29].

Qi et al. [171], [172] empirically compared the effectiveness of 14 spectrum-based fault localization techniques in terms of the number of invalid candidate solutions that GenProg produces before generating a valid fix. In this

experiment, the Jaccard coefficient [90] always results in higher or equal effectiveness compared to other localization techniques.

Automatic bug fixing is cheap: Automatic software repairing is not for free because fixes require significant resources to be generated. Two studies quantified this cost for the AE and GenProg search-based techniques [87], [111]. They conducted the experiments on the Amazon's cloud computing infrastructure, and they computed the cost per fault by dividing the cost of renting the infrastructure by the number of bugs that have been repaired. Early results show that the cost of fixing a fault is 4.40$ with AE and 14.78$ with GenProg.

Although automatically fixing faults seem to be extremely cheap, this evaluation does not consider the effort that the developers have to spend inspecting the output produced by software repair techniques to check the fixes. Note that several fixes might be discarded by the developers [110], so doing this check is not a trivial activity. Studying the overall cost of the automatic repair process is still an open issue.

The representation of the problem (P_{repair}) has a strong impact on the effectiveness of the repair process: Generate-and-validate techniques differ in the representation of the individuals, the definition of the search space, and the design of the mutation and crossover operators. Le Goues et al. [160] empirically studied the impact of various aspects on the repair process. One of the reported results shows that representing individuals as a list of modifications to the original program outperforms AST representation in terms of success rate.

Smoothening the gradient of the fitness function may potentially improve the automatic repair process. Fast et al. [163] investigated the idea of using Daikon [16] to learn program invariants that characterize the behavior of successful and failed executions. The differences between the program invariants associated with the program under repair and the invariants associated with the candidate solutions can be used in the definition of the fitness function, smoothening its gradient: a specific candidate solution might still fail a test case but it could turn an invariant to true. In this way the repair process may better capture progresses toward the identification of a fix. In their experiments, Fast et al. observed that a smoothed fitness function can better guide the search process, but the cost of computing this specific function overcame the benefits for the search process overall resulting in a 3 percent average overhead.

8.3 Evidence for Semantics-Driven Approaches

Semantics-driven techniques have been studied less extensively than generate-and-validate techniques and many of the factors that influence their effectiveness are still unknown. However, the empirical results collected so far are sufficient to report some interesting observations.

The size of the test suite may strongly influence the generation of the fixes: As for the generate-and-validate approaches, bigger test suites produce more data making the generation of the fixes harder due to the many constraints that must be satisfied to match the collected data. For instance, the problem of synthesizing a new clause for an *if* condition is harder if the new clause has to satisfy thousands of constraints collected by executing a large test suite rather then a dozen constraints. Intuitively it is harder to generate repairs that pass many tests. However, it is important to note that fixes that are generated from small test suites might be inaccurate and produce wrong behaviors when thoroughly validated. It is thus important also for semantics-driven techniques to work with test suites that extensively cover the behavior of the program for the statements that must be repaired, without being unnecessarily large or redundant, as early experienced in SemFix [21].

Program synthesis techniques are faster than expression enumeration techniques: Identifying a fix by explicitly enumerating all the possible expressions that can be constructed starting from a set of program variables and operators is easier than using program synthesis techniques, but it is also significantly slower. For instance, SemFix estimates that producing constraints by enumeration might be four times slower than using program synthesis techniques [21].

SMT solving is the dominating cost factor in the repair process: While test case execution is the major cost factor in generate-and-validate techniques, SMT solving is the dominating cost factor in semantics-driven techniques. It has been reported that SMT solving can take up to 99 percent of the total repair time [26]. Designing semantics-driven techniques that make a better use of solvers is thus an important direction for the future.

Fixes of concurrency faults may significantly affect the performance of the program under repair: Concurrency fault detection techniques might be inaccurate, for instance identifying large code regions as faulty regions (e.g., regions that may trigger an atomicity violation bug). As a consequence, repair techniques may generate large critical regions protected by lock and unlock operations producing significant overhead in the program under repair [82]. This is an example of how a fix for a fault (e.g., a concurrency fault) might introduce another fault (e.g., a performance fault).

9 Open Challenges

Automatic program repair techniques have already demonstrated their effectiveness in a number of non-trivial contexts, but their general applicability as well as the possibility to exploit this technology in industrial settings must still be demonstrated. In our analysis, we identified a number of important challenges as well as research trends suggesting where the future effort in the area might concentrate. In this section we report these challenges organized in three sections: fix correctness challenges, process challenges, and technical challenges.

9.1 Fix Correctness Challenges

Plausible versus Correct Solutions: Most of the techniques can generate *plausible* fixes, for instance fixes that pass all the test cases in an available test suite. In contrast, developers need *correct* fixes, that is, fixes that satisfy all the requirements of the program under repair. Although the use of test suites has undoubtedly been the main method to validate the automatically generated fixes, other approaches have

been also experienced, such as the use of specifications [42] and code contracts [94]. However, also in these cases the generated fixes are only plausible, that is, they are correct according to the specific validation criterion that has been used (e.g., a specification or a contract), and they are not necessarily correct for the developers.

To mitigate this issue, some approaches investigated the definition of repair operators inspired from actual fixes written by developers [22], [119], [120], others considered the manual inspection of the fixes as the most useful validation method [55]. However, it is still a generally open challenge to identify methods to produce fixes that are correct and understandable according to the developers' judgment [51]. Although it might be infeasible to produce fixes that are ''guaranteed'' to satisfy the developers' expectations, finding methods to generate and evaluate fixes that are *likely* acceptable by developers is an important research direction.

Non-Functional Qualities of the Fixes: The gap between plausible and correct solutions is exacerbated by the many non-functional qualities that a fix should satisfy to be fully acceptable by a software developer. For instance, developers may need fixes that are both secure and computationally efficient, aspects that are often badly covered or even ignored by the available test suites. The challenge of satisfying the non-functional qualities, especially the ones that are hard to test with test suites, is an open research issue.

The clarity of the fix is also important. An acceptable program change must usually satisfy structural and syntactical requirements defined in terms of best coding practices and naming conventions. These practices may vary a lot organization by organization, and automatic repair techniques must satisfy a non-trivial level of customization and sophistication to effectively consider all these aspects when producing fixes.

Cross-Validating the Automatically Generated Fixes. Although a fix produced by an automatic program repair technique might be directly accepted without any modification by a developer, in practice developers need to check its correctness before it can be integrated into a program. While some fixes might be trivial to check, a number of fixes might be non-obvious and require a certain degree of adaptation to be finalized. The effort required to cross-check and understand a fix might be significant, potentially mitigating the benefit of automatic techniques.

However automatic program repairing techniques, in addition to generating fixes, might be able to generate information that explains the rationale of the fix and that facilitates and assists the developer during the validation of the fix. Producing this additional piece of knowledge is an open challenge.

9.2 Process Challenges

Selecting the right technique for the right case: How to select the appropriate program repair solution in a given situation is an important open problem. There are choices to be made at multiple levels. On the first place, developers may prefer fully automatic repair techniques to fix recommenders, or vice versa. This choice may depend on the developer's preferences, but also on the nature of the repair task. For instance, some tasks might be hard to be completely automated and fix recommenders may represent a valid option in these cases. Unfortunately, there is still little understanding when one class of approaches should be preferred to the other.

Another variable is whether opting for a fault-specific or a general technique. A good strategy might be preferring fault-specific techniques when applicable, and use general techniques otherwise. Unfortunately it might be hard to guess a-priori what the fault to be repaired is, and thus it might be extremely difficult to choose the right fault-specific technique, unless the failure is already explicative of the nature of the fault, such as for infinite loop failures.

Another level of decision is the choice of the type of change model and of the algorithmic strategy that should be adopted to fix a fault. The ability to make these choices might significantly facilitate the selection of a repair technique. However, since a fault is unknown until it has been fixed, taking these decisions is extremely hard in practice.

A preliminary result about the identification of the method that should be used to address a fault is the work by Le et al. who defined a method to decide if a bug report should be addressed with standard techniques or with an automatic program repair technique [173].

Defining Methodologies and Adapting Software Processes: The definition of suitable methodologies and software processes to efficiently integrate software repair techniques into industrial processes is still a completely open research area. Aspects that deserve more attention are understanding how to design programs that can be effectively repaired automatically, defining when automatic program repair routines should be executed and how their output should be integrated into organization processes, and revise responsibilities and roles of the quality assurance process.

The Challenge of a Fully Automatic Repair Process: While program repair techniques can sometime automatically produce reasonable fixes, the acceptability of a fix is still relegated to the judgment of developers. The benefit of using these techniques would be evident if the repair process could be fully automated, at least in some cases. Some approaches investigated the possibility to automatically generate and deploy ''quick-and-dirty'' fixes in the field [101], [174], [175], but automatically producing developer-quality fixes is still a long-term challenge.

Program Maintainability: While it is well-known how to maintain programs subject to regular human-driven changes, how to maintain software programs subject to a mix of changes actuated by both developers and automatic program repairing techniques is still unclear. In principle, automatic fixes of low internal quality may hinder code maintainability, and their accumulation may lead to major issues on software evolvability. Designing techniques that fix programs without negatively affecting the maintainability of a software project is still a largely unexplored area.

9.3 Technical Challenges

Scattered Set of Findings: There are many software repair techniques and many studies available, however findings are still scattered and difficult to be synthesized into a clear picture. This is often due to the mix of innovation (e.g., presenting a new strategy for the synthesis of program fixes), and optimizations (e.g., prioritizing the test cases for the validation) introduced by each work in the area. Due to the

overlapping effect of these factors it is sometime hard to establish what the factor influencing more the results is. Is it the strategy? Is it a specific heuristic? Is it the quality of the implementation? The way all these factors may interfere when comparing and studying the solutions in a same area has been also discussed in [176].

It is important to improve the maturity of the field and obtain a better understanding of the strategies and the heuristics that are important and useful, and that should be present in every automatic program repair technique, for instance by increasing the number of independent studies on the impact of equivalence checking [111], optimized representation of the candidate solutions [27], [102], and test prioritization schema [29], [106].

Test Suites for Program Repair: There is clear evidence that test suites may have a dramatic impact on the effectiveness of both generate-and-validate (see Section 8.2.1) and semantics-driven techniques (see Section 8.3). However there is only initial evidence of the factors that characterize a good test suite for program repair, such as the strength of the oracle [110] and its completeness [36], [159]. This results in the lack of guidelines about the construction of test suites that can optimally support the automatic repair process. Defining proper methods to create these test suites may have a dramatic impact on the effectiveness of program repair solutions.

Design for Program Repair: There is evidence that the complexity and the structure of a program (e.g., the presence of statements that may increase the effectiveness of general program repair techniques [100], [167]) as well as the presence of annotations (e.g., the presence of function pre- and post-conditions [92]) have an influence on the success and the type of program repair techniques that can be applied. This raises a more general question about whether we can build software that facilitates program repairing, for instance using a specific coding style, or avoiding/preferring some programming constructs to others, or using specific libraries. Since we cannot expect program repair techniques to be able to address every possible situation, we should think about building software ready to be repaired automatically.

Alternative Validation Mechanisms: Almost all the generate-and-validate techniques use testing as validation mechanism. However, test cases have demonstrated several limitations when used to validate candidate solutions. In particular, a weak test suite may favour repairs that drop functionalities, and strong test suites may make the generation of correct fixes extremely hard [110]. While understanding how to design good test suite for program repair is definitely an option, test suites are not the only option for validation. For instance, FixMeUp [42] uses access control rules as validation mechanism. More in general, automatic program repair should investigate alternative validation mechanisms, for instance based on the reuse of the same artifacts used to define the requirements or the design of the software, that might complement the validation performed with the test suite obtaining better fixes.

Producing Better Search Spaces: The search spaces defined so far have been demonstrated to not include enough correct programs to enable the design of efficient and general program repair solutions [159]. It is thus necessary to produce better search spaces with a higher density of correct fixes that can be efficiently discovered with program repair techniques. To this end, it is important to exploit as many sources of information as possible, including fixes from other applications and systems, user hints, specifications, and comments. So far, only the existing fixes have been exploited by example-based techniques (see Section 6.1.3).

Benchmarks: To let the field progress, it is important to have a good number of benchmarks that researchers could use to assess and compare techniques. Automatic program repair techniques have already extensively exploited some of the available benchmarks, such as the applications and the faults defined in the ManyBugs benchmark [154] and in the SIR repository [156]. While these benchmarks are definitely useful, it is important to better exploit the available resources and define additional benchmarks enriching the set of faults and cases considered as subjects of the experiments to prevent the risk of collecting empirical evidence biased by the use of a limited set of benchmark applications and faults, and to increase the capability of delivering findings that generalize beyond the specific cases that have been analyzed.

Better Use of Test Suites and Solvers: Running the test suites in generate-and-validate techniques [41] and running the solvers in semantics-driven techniques [26] have been reported as the dominating cost factors. Since the design of more efficient solutions allows a more thorough exploration of the search space and a more effective synthesis, it is important to reduce these costs. Some techniques have partially addressed this problem for instance prioritizing the test cases to reduce the cost of test execution [29], [106]. However, it is necessary to design repair techniques that perform a definitely smarter and better use of these expensive technologies to improve program repair solutions by order of magnitudes.

10 Conclusions

Automatic program repair techniques address the ambitious challenge of automatically repairing faulty software. In the last decade, program repair techniques have produced relevant and impactful results demonstrating the potential of significantly affecting testing, validation, and debugging practices on the long term.

This paper organizes the knowledge in the area, surveying the existing work, and discussing the main results and challenges. In particular, this paper shows that program repairing approaches address the problem of repairing software according to two main strategies. Generate-and-validate approaches generate a search space of the possible fixes and then explore this search space looking for a correct fix. Semantics-driven approaches produce a representation of the problem of fixing a program and solve this problem to obtain the actual fixes.

We integrate and summarize the empirical evidence collected from multiple diverse studies into a set of key facts that show the tradeoffs and factors influencing the effectiveness of these approaches. We finally discuss the challenges we believe are the most important and that can influence the future research in the area.

APPENDIX A
PUBLICLY AVAILABLE TOOLS

Technique	Tool	Language
Angelix	http://angelix.io/	C
ARC	https://github.com/sqrlab/arc/	Java
AutoFix-E2 (AutoFix)	http://se.inf.ethz.ch/research/autofix/	Eiffel
ConcBugAssist	http://www-bcf.usc.edu/~wang626/project_concbugassist.htm	C
DFixer	http://lcs.ios.ac.cn/~yancai/dfixer/release_lk/SourceCodeRelease_2015Nov17.7z	C
DynaMoth	https://github.com/SpoonLabs/nopol	Java
AE	http://dijkstra.cs.virginia.edu/genprog/	C
GenProg	http://dijkstra.cs.virginia.edu/genprog/	C
Grail	https://github.com/lpxz?tab=repositories	Java
History-driven repair	https://github.com/xuanbachle/bugfixes	Java
JAFF	http://sachaproject.gforge.inria.fr/mirror/abf-jaff-arcuri.zip	Java
KALI	http://groups.csail.mit.edu/pac/patchgen/	C
Leakfix	http://sei.pku.edu.cn/~gaoqing11/leakfix/	C
MintHint	http://www.iisc-seal.net/minthint	C
MUT-APR	http://fyassiri.wixsite.com/mutapr	C
Nopol	https://github.com/SpoonLabs/nopol/	Java
PACHIKA	http://www.st.cs.uni-saarland.de/models/	Java
Prophet	http://groups.csail.mit.edu/pac/patchgen/	C
pyEDB	https://bitbucket.org/tomackling/pyedb	Python
QACrashFix	http://sei.pku.edu.cn/~gaoqing11/qacrashfix/	Java
RSRepair (aka TrpAutoRepair)	http://qiyuhua.github.com/projects/rsrepair/	C
SearchRepair	https://github.com/ProgramRepair/SearchRepair/	C
SemRep	http://cs.ucsb.edu/~vlab/tools.html	php
SPR	http://groups.csail.mit.edu/pac/patchgen/	C
Surendran et al.	http://dl.acm.org/citation.cfm?id=2594335	Habanero Java

ACKNOWLEDGMENTS

The authors would like to thank Fan Long, Shan Lu, Xiaoguang Mao, and Abhik Roychoudhury for their comments on an early version of this paper. This work has been partially supported by the EU H2020 ''Learn'' project, which has been funded under the ERC Consolidator Grant 2014 program (ERC Grant Agreement n. 646867) and the ''GAUSS'' national research project, which has been funded by the MIUR under the PRIN 2015 program (Contract 2015KWREMX).

REFERENCES

[1] T. Britton, L. Jeng, G. Carver, and P. Cheak, ''Reversible debugging software - quantify the time and cost saved using reversible debuggers,'' 2013.

[2] Undo Software, ''Increasing software development productivity with reversible debugging,'' Undo Software, Tech. Rep. white paper, 2014.

[3] M. A. Müllerburg, ''The role of debugging within software engineering environments,'' *ACM SIGSOFT Softw. Eng. Notes*, vol. 8, no. 4, pp. 81–90, 1983, doi: 10.1145/1006147.1006165.

[4] B. Hailpern and S. Padmanabhan, ''Software debugging, testing, and verification,'' *IBM Syst. J.*, vol. 41, no. 1, pp. 4–12, 2002, doi: 10.1147/sj.411.0004.

[5] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. Burlington, MA, USA: Morgan Kaufmann, 2009.

[6] J. A. Jones and M. J. Harrold, ''Empirical evaluation of the Tarantula automatic fault-localization technique,'' in *Proc. Int. Conf. Automated Softw. Eng.*, 2005, pp. 273–282, doi: 10.1145/1101908.1101949.

[7] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, ''Sober: Statistical model-based bug localization,'' in *Proc. Joint Meet. Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2005, pp. 286–295, doi: 10.1145/1081706.1081753.

[8] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, ''Spectrum-based multiple fault localization,'' in *Proc. Int. Conf. Automated Softw. Eng.*, 2009, pp. 88–99, doi: 10.1109/ASE.2009.25.

[9] A. Zeller, ''Yesterday, my program worked. Today, it does not. Why?'' in *Proc. Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 1999, pp. 253–267, doi: 10.1007/3-540-48166-4_16.

[10] A. Zeller and R. Hildebrandt, ''Simplifying and isolating failure-inducing input,'' *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002, doi: 10.1109/32.988498.

[11] A. Zeller, ''Isolating cause-effect chains from computer programs,'' in *Proc. Symp. Foundations Softw. Eng.*, 2002, pp. 1–10, doi: 10.1145/587051.587053.

[12] L. Mariani and F. Pastore, ''Automated identification of failure causes in system logs,'' in *Proc. Int. Symp. Softw. Rel. Eng.*, 2008, pp. 117–126, doi: 0.1109/ISSRE.2008.48.

[13] A. Babenko, L. Mariani, and F. Pastore, ''AVA: Automated interpretation of dynamically detected anomalies,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2009, pp. 237–248, doi: 10.1145/1572272.1572300.

[14] L. Mariani, F. Pastore, and M. Pezzè, ''Dynamic analysis for diagnosing integration faults,'' *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 486–508, Jul./Aug. 2011, doi: 10.1109/TSE.2010.93.

[15] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, ''MIMIC: locating and understanding bugs by analyzing mimicked executions,'' in *Proc. Int. Conf. Automated Softw. Eng.*, 2014, pp. 815–826, 10.1145/2642937.2643014.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, ''Dynamically discovering likely program invariants to support program evolution,'' *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001, doi: 10.1145/302405.302467.

[17] D. Lorenzoli, L. Mariani, and M. Pezzè, ''Automatic generation of software behavioral models,'' in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 501–510, doi: 10.1145/1368088.1368157.

[18] M. Gabel and Z. Su, ''Testing mined specifications,'' in *Proc. Int. Symp. Foundations Softw. Eng.*, 2012, pp. 4:1–4:11, doi: 10.1145/2393596.2393598.

[19] I. Krka, Y. Brun, and N. Medvidovic, ''Automatic mining of specifications from invocation traces and method invariants,'' in *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 178–189, doi: 10.1145/2635868.2635890.

[20] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, ''Automatically finding patches using genetic programming,'' in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 364–374, doi: 10.1109/ICSE.2009.5070536.

[21] H. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, ''Semfix: Program repair via semantic analysis,'' in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 772–781, doi:10.1109/ICSE.2013.6606623.

[22] D. Kim, J. Nam, J. Song, and S. Kim, ''Automatic patch generation learned from human-written patches,'' in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 802–811, doi: 10.1109/ICSE.2013.6606626.

[23] F. DeMarco, J. Xuan, D. L. Berre, and M. Monperrus, ''Automatic repair of buggy if conditions and missing preconditions with SMT,'' in *Proc. Int. Workshop Constraints Softw. Testing Verification Anal.*, 2014, pp. 30–39, doi: 10.1145/2593735.2593740.

[24] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, ''Automatic error elimination by horizontal code transfer across multiple applications,'' in *Proc. Conf. Program. Language Des. Implementation*, 2015, pp. 43–54, doi: 10.1145/2737924.2737988.

[25] V. Dallmeier, A. Zeller, and B. Meyer, ''Generating fixes from object behavior anomalies,'' in *Proc. Int. Conf. Automated Softw. Eng.*, 2009, pp. 550–554, doi: 10.1109/ASE.2009.15.

[26] S. Marcote and M. Monperrus, ''Automatic repair of infinite loops,'' Tech. Rep. hal-01144026, 2015.

[27] T. Ackling, B. Alexander, and I. Grunert, ''Evolving patches for software repair,'' in *Proc. Annu. Conf. Genetic Evol. Comput.*, 2011, pp. 1427–1434, doi: 10.1145/2001576.2001768.

[28] F. Long and M. Rinard, ''Staged program repair with condition synthesis,'' in *Proc. Joint Meet. Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2015, pp. 166–178, doi: 10.1145/2786805.2786811.

[29] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 254–265, doi: 10.1145/2568225.2568254.

[30] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 64–74, doi: 10.1145/2635868.2635873.

[31] W. Weimer, "Patches as better bug reports," in *Proc. Int. Conf. Generative Program. Component Eng.*, 2006, pp. 181–190, doi: 10.1145/1173706.1173734.

[32] A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, "On the evaluation of automatic program repair techniques and tools," in *Proc. Iranian Conf. Elect. Eng.*, 2016, pp. 61–66, doi: 10.1109/IranianCEE.2016.7585390.

[33] M. Monperrus, "Automatic software repair: A bibliography," Univ. Lille, Lille, France, Tech. Rep. hal-01206501, 2015.

[34] M. Monperrus and B. Baudry, "Two flavors in automated software repair: Rigid repair and plastic repair," in *Dagstuhl Seminar n. 13061 "Fault Prediction, Localization, and Repair"*, 2013.

[35] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. World Congr. Comput. Intell.*, 2008, pp. 162–168.

[36] F. Assiri and J. Bieman, "An assessment of the quality of automated program operator repair," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2014, pp. 273–282, doi: 10.1109/ICST.2014.40.

[37] J. Bradbury and K. Jalbert, "Automatic repair of concurrency bugs," in *Proc. Int. Symp. Search Based Softw. Eng.*, 2010, pp. 73–84.

[38] D. Kelk, K. Jalbert, and J. S. Bradbury, "Automatically repairing concurrency bug with ARC," in *Proc. Int. Conf. Multicore Softw. Eng., Perform. Tools*, 2013, pp. 73–84, doi: 10.1007/978-3-642-39955-8_7.

[39] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "Bugfix: A learning-based tool to assist developers in fixing bugs," in *Proc. Int. Conf. Program Comprehension*, 2009, pp. 70–79, doi: 10.1109/ICPC.2009.5090029.

[40] A. Arcuri and X. Yao, "Coevolving programs and unit tests from their specification," in *Proc. Int. Conf. Automated Softw. Eng.*, 2007, pp. 397–400, doi: 10.1145/1321631.1321693.

[41] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Springer Softw. Quality J.*, vol. 21, no. 3, pp. 421–443, 2013, doi: 10.1007/s11219-013-9208-0.

[42] S. Son, K. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications," in *Proc. Netw. Distrib. Syst. Symp.*, 2013.

[43] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Proc. Int. Conf. Comput. Aided Verification*, 2005, pp. 226–238, doi: 10.1007/11513988_23.

[44] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proc. Int. Conf. Automated Softw. Eng.*, 2015, pp. 295–306, doi: 10.1109/ASE.2015.60.

[45] V. Debroy and E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2010, pp. 65–74, doi: 10.1109/ICST.2010.66.

[46] M. Stumptner and F. Wotawa, "A model-based approach to software debugging," in *Proc. Int. Workshop Principles Diagnosis*, 1996, pp. 233–239, doi: 10.1007/978-3-540-92814-0_36.

[47] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," in *Proc. Adv. Res. Work. Conf. Correct Hardware Des. Verification Methods*, 2005, pp. 35–49, doi: 10.1007/11560548_6.

[48] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing model checking in verification by AI techniques," *Artif. Intell.*, vol. 112, no. 1–2, pp. 57–104, 1999, doi: 10.1016/S0004-3702(99)00039-9.

[49] A. Dennis, R. Monroy, and P. Nogueira, "Proof-directed debugging and repair," in *Proc. Symp. Trends Functional Program.*, 2006, pp. 131–140.

[50] J. Wilkerson, D. Tauritz, and J. Bridges, "Multi-objective coevolutionary automated software correction," in *Proc. Annu. Conf. Genetic Evol. Comput.*, 2012, pp. 1229–1236, doi: 10.1145/2330163.2330333.

[51] Z. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 177–187, doi: 10.1145/2338965.2336775.

[52] F. Yu, C.-Y. Shueh, C.-H. Lin, Y.-F. Chen, B.-Y. Wang, and T. Bultan, "Optimal sanitization synthesis for web application vulnerability repair," in *Proc. Int. Symp. Softw. Testing Anal.*, 2016, pp. 189–200, doi: 10.1145/2931037.2931050.

[53] Y. Cai and L. Cao, "Fixing deadlocks via lock pre-acquisitions," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 1109–1120, doi: 10.1145/2884781.2884819.

[54] Q. Gao, et al., "Safe memory-leak fixing for C programs," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 459–470, doi: 10.1109/ICSE.2015.64.

[55] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing Q&A sites," in *Proc. Int. Conf. Automated Softw. Eng.*, 2015, pp. 307–318, doi: 10.1109/ASE.2015.81.

[56] A. Nistor, P. Chang, C. Radoi, and S. Lu, "CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 902–912, doi: 10.1109/ICSE.2015.100.

[57] A. W. Biermann, "Automatic programming: A tutorial on formal methodologies," *J. Symbolic Comput.*, vol. 1, no. 2, pp. 119–142, 1985, doi: http://dx.doi.org/10.1016/S0747-7171(85)80010-9.

[58] D. L. Parnas, "Software aspects of strategic defense systems," *Commun. ACM*, vol. 28, no. 12, pp. 1326–1335, 1985, doi: 10.1145/214956.214961.

[59] O. Polozov and S. Gulwani, "FlashMeta: A framework for inductive program synthesis," in *Proc. Int. Conf. Object-Oriented Program. Syst. Languages Appl.*, 2015, pp. 107–126, doi: 10.1145/2814270.2814310.

[60] R. Alur, et al., "Syntax-guided synthesis," in *Proc. Int. Conf. Formal Methods Comput.-Aided Des.*, 2013, pp. 1–8, doi: 10.1109/FMCAD.2013.6679385.

[61] V. Le and S. Gulwani, "FlashExtract: A framework for data extraction by examples," in *Proc. Conf. Program. Language Des. Implementation*, 2014, pp. 542–553, doi: 10.1145/2594291.2594333.

[62] M. Stumptner and F. Wotawa, "Model-based program debugging and repair," in *Proc. Int. Conf. Ind., Eng. Other Appl. Appl. Intell. Syst.*, 1996, pp. 155–160.

[63] A. Arcuri, "On the automation of fixing software bugs," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 1003–1006, doi: 10.1145/1370175.1370223.

[64] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proc. Annu. Conf. Genetic Evol. Comput.*, 2009, pp. 947–954, doi: 10.1145/1569901.1570031.

[65] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, pp. 109–116, 2010, doi: 10.1145/1735223.1735249.

[66] H. Psaier and S. Dustdar, "A survey on self-healing systems: approaches and systems," *Springer Comput.*, vol. 91, no. 1, pp. 43–73, 2011, doi: 10.1007/s00607-010-0107-y.

[67] D. Ghosh, R. Sharman, H. Rao, and S. Upadhyaya, "Self-healing systems - survey and synthesis," *Elsevier Decision Support Syst.*, vol. 42, no. 4, pp. 2164–2185, 2007, doi: 10.1016/j.dss.2006.06.011.

[68] A. Keromytis, "Characterizing self-healing software systems," in *Proc. Int. Conf. Math. Methods Models Archit. Comput. Netw. Secur*, 2007, pp. 22–33, doi: 10.1007/978-3-540-73986-9_2.

[69] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds: Exploiting the intrinsic redundancy of web applications," *ACM Trans. Softw. Eng. Methodologies*, vol. 24, no. 3, pp. 16:1–16:42, 2015, doi: 10.1145/2755970.

[70] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Hoboken, NJ, USA: Wiley, 2007.

[71] M. Carbin, S. Misailovic, M. Kling, and M. Rinard, "Detecting and escaping infinite loops with jolt," in *Proc. Eur. Conf. Object-Oriented Program.*, 2011, pp. 609–633, doi: 10.1007/978-3-642-22655-7_28.

[72] O. Riganelli, D. Micucci, and L. Mariani, "Policy enforcement with proactive libraries," in *Proc. Int. Symp. Softw. Eng. Adaptive Self-Manag. Syst.*, 2017, pp. 182–192, doi: 10.1109/SEAMS.2017.9.

[73] D. G. De La Iglesia and D. Weyns, "MAPE-K formal templates to rigorously design behaviors for self-adaptive systems," *ACM Trans. Autonomous and Adaptive Syst.*, vol. 10, no. 3, pp. 15:1–15:31, 2015, doi: 10.1145/2724719.

[74] R. Ding, et al., "Healing online service systems via mining historical issue repositories," in *Proc. Int. Conf. Automated Softw. Eng.*, 2012, pp. 318–321, doi: 10.1145/2351676.2351735.

[75] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 782–791, doi: 10.1109/ICSE.2013.6606624.

[76] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin, *A Practical Guide to the IBM Autonomic Computing Toolkit*. Armonk, NY, USA: IBM Redbooks, 2004.

[77] S. Colin and L. Mariani, ''Run-time verification,'' in *Model-Based Testing of Reactive Systems*, Berlin, Germany: Springer, 2005, vol. 3472, pp. 525–555, doi: 10.1007/11498490_24.

[78] E. T. Barr, M. Harman, P. McMinn, and M. Shahbaz, ''The oracle problem in software testing: A survey,'' *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015, doi: 10.1109/TSE.2014.2372785.

[79] Y. Zhang and A. Mesbah, ''Assertions are strongly correlated with test suite effectiveness,'' in *Proc. Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2015, pp. 214–224, doi: 10.1145/2786805.2786858.

[80] H. Chang, L. Mariani, and M. Pezzè, ''Exception handlers for healing component-based systems,'' *ACM Trans. Softw. Eng. Methodology*, vol. 22, no. 4, pp. 30:1–30:40, 2013, doi: 10.1145/2522920.2522923.

[81] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, ''Assure: automatic software self-healing using rescue points,'' *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 37–48, 2009, doi: 10.1145/1508284.1508250.

[82] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, ''Automated atomicity-violation fixing,'' *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 389–400, 2011, doi: 10.1145/1993498.1993544.

[83] S. Park, S. Lu, and Y. Zhou, ''CTrigger: Exposing atomicity violation bugs from their hiding places,'' in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2009, pp. 25–36, doi: 10.1145/1508244.1508249.

[84] P. Agarwal and A. Agrawal, ''Fault-localization techniques for software systems: A literature review,'' *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 1–8, 2014, doi:10.1145/2659118.2659125.

[85] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, ''Scalable statistical bug isolation,'' in *Proc. Conf. Program. Language Des. Implementation*, 2005, pp. 15–26, doi: 10.1145/1065010.1065014.

[86] W. E. Wong, V. Debroy, and B. Choi, ''A family of code coverage-based heuristics for effective fault localization,'' *J. Syst. Softw.*, vol. 83, no. 2, pp. 188–208, 2010, doi:10.1016/j.jss.2009.09.037.

[87] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, ''A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,'' in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 3–13, doi: 10.1109/ICSE.2012.6227211.

[88] C. Le Goues, ''Automatic program repair using genetic programming,'' Ph.D. dissertation, Faculty School Eng. Appl. Sci., Univ. Virginia, Charlottesville, VA, USA, 2013.

[89] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, ''On the accuracy of spectrum-based fault localization,'' in *Proc. Testing: Academic Ind. Conf. Practice Res. Techn. - MUTATION*, 2007, pp. 89–98, doi: 10.1109/TAIC.PART.2007.13.

[90] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, ''Pinpoint: Problem determination in large, dynamic internet services,'' in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 595–604, doi: 10.1109/DSN.2002.1029005.

[91] Y. Qi, X. Mao, Y. Lei, and C. Wang, ''Using automated program repair for evaluating the effectiveness of fault localization techniques,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 191–201, doi: 10.1145/2483760.2483785.

[92] Y. Wei, et al., ''Automated fixing of programs with contracts,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2010, pp. 61–72, doi: 10.1145/1831708.1831716.

[93] Y. Pei, Y. Wei, C. Furia, M. Nordio, and B. Meyer, ''Code-based automated program fixing,'' in *Proc. Int. Conf. Automated Softw. Eng.*, 2011, pp. 392–395, doi: 10.1109/ASE.2011.6100080.

[94] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, ''Automated fixing of programs with contracts,'' *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 427–449, May 2014, doi: 10.1109/TSE.2014.2312918.

[95] S. Mechtaev, J. Yi, and A. Roychoudhury, ''Angelix: Scalable multiline program patch synthesis via symbolic analysis,'' in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 691–701, doi: 10.1145/2884781.2884807.

[96] R. Kou, Y. Higo, and S. Kusumoto, ''A capable crossover technique on automatic program repair,'' in *Proc. Int. Workshop Empirical Softw. Eng. Practice*, 2016, pp. 45–50, doi: 10.1109/IWESEP.2016.15.

[97] J. Wilkerson and D. Tauritz, ''Coevolutionary automated software correction,'' in *Proc. Annu. Conf. Genetic and Evol. Comput.*, 2010, pp. 1391–1392, doi: 10.1145/1830483.1830739.

[98] T. Ji, L. Chen, X. Mao, and X. Yi, ''Automated program repair by using similar code containing fix ingredients,'' in *Proc. Annu. Comput. Softw. Appl. Conf.*, 2016, pp. 197–202, doi: 10.1109/COMPSAC.2016.69.

[99] A. Arcuri, ''Evolutionary repair of faulty software,'' *Appl. Soft Comput.*, vol. 11, no. 4, pp. 3494–3514, 2011, doi: 10.1016/j.asoc.2011.01.023.

[100] E. T. Barr, T. Brun, P. Devanbu, M. Harman, and F. Sarro, ''The plastic surgery hypothesis,'' in *Proc. Int. Symp. Foundations Softw. Eng.*, 2014, pp. 306–317, doi: 10.1145/2635868.2635898.

[101] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, ''GenProg: A generic method for automatic software repair,'' *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012, doi: 10.1109/TSE.2011.104.

[102] Y. Qi, X. Mao, and Y. Lei, ''Making automatic repair for large-scale programs more efficient using weak recompilation,'' in *Proc. Int. Conf. Softw. Maintenance*, 2012, pp. 254–263, doi: 10.1109/ICSM.2012.6405280.

[103] Y. Qi, X. Mao, Z. Dai, and Y. Qi, ''Efficient automatic program repair using function-based part-execution,'' in *Proc. Int. Conf. Softw. Eng. Service Sci.*, 2013, pp. 235–238, doi: 10.1109/ICSESS.2013.6615295.

[104] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, ''Automated repair of binary and assembly programs for cooperating embedded devices,'' *ACM SIGPLAN Notices*, vol. 48, no. 4, 2013, pp. 317–328, doi: 10.1145/2499368.2451151.

[105] E. Schulte, S. Forrest, and W. Weimer, ''Automated program repair through the evolution of assembly code,'' in *Proc. Int. Conf. Autom. Softw. Eng.*, 2010, pp. 313–316, doi: 10.1145/1858996.1859059.

[106] Y. Qi, X. Mao, and Y. Lei, ''Efficient automated program repair through fault-recorded testing prioritization,'' in *Proc. Int. Conf. Softw. Maintenance*, 2013, pp. 180–189, doi: 10.1109/ICSM.2013.29.

[107] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, ''Prioritizing test cases for regression testing,'' *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001, doi: 10.1109/32.962562.

[108] J. Wilkerson and D. Tauritz, ''Scalability of the coevolutionary automated software correction system,'' in *Proc. Conf. Companion Genetic Evol. Comput.*, 2011, pp. 243–244, doi: 10.1145/2001858.2001995.

[109] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, ''Mining object behavior with ADABU,'' in *Proc. Int. Workshop Dynamic Anal.*, 2006, pp. 17–24, doi: 10.1145/1138912.1138918.

[110] Z. Qi, F. Long, S. Achour, and M. Rinard, ''An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36, doi: 10.1145/2771783.2771791.

[111] W. Weimer, Z. Fry, and S. Forrest, ''Leveraging program equivalence for adaptive program repair: Models and first results,'' in *Proc. Int. Conf. Autom. Softw. Eng.*, 2013, pp. 356–366, doi: 10.1109/ASE.2013.6693094.

[112] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, ''Multithreaded Java program test generation,'' *IBM Syst. J.*, vol. 41, no. 1, pp. 111–125, 2002, doi: 10.1147/sj.411.0111.

[113] F. Long and M. Rinard, ''Automatic patch generation by learning correct code,'' in *Proc. Annu. Symp. Principles Program. Languages*, 2016, pp. 298–312, doi: 10.1145/2914770.2837617.

[114] F. Long and M. Rinard, ''Prophet: Automatic patch generation via learning from successful patches,'' MIT, Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2015-027, 2013.

[115] A. Smirnov and T. Chiueh, ''Automatic patch generation for buffer overflow attacks,'' in *Proc. Int. Symp. Inform. Assurance Secur.*, 2007, pp. 165–170, doi: 10.1109/IAS.2007.87.

[116] T. Chiueh and F. Hsu, ''RAD: A compile-time solution to buffer overflow attacks,'' in *Proc. Int. Conf. Distributed Comput. Syst.*, 2001, pp. 409–417, doi: 10.1109/ICDSC.2001.918971.

[117] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, ''Autopag: Towards automated software patch generation with source code root cause identification and repair,'' in *Proc. Symp. Inform., Comput. Commun. Secur.*, 2007, pp. 329–340, doi: 10.1145/1229285.1267001.

[118] X.-B. D. Le, D. Lo, and C. Le Goues, ''History driven program repair,'' in *Proc. Int. Conf. Softw. Anal. Evol. and Reengineering*, 2016, pp. 213–224, doi: 10.1109/SANER.2016.76.

[119] S. Tan and A. Roychoudhury, ''Relifix: Automated repair of software regressions,'' in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 471–482, doi: 10.1109/ICSE.2015.65.

[120] C. Liu, J. Yang, L. Tan, and M. Hafiz, ''R2fix: Automatically generating bug fixes from bug reports,'' in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 282–291, doi: 10.1109/ICST.2013.24.

[121] M. Monperrus, ''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 *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 234–242, doi: 10.1145/2568225.2568324.

[122] X.-B. D. Le, ''Towards efficient and effective automatic program repair,'' in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 876–879, doi: 10.1145/2970276.2975934.

[123] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, ''An experimental determination of sufficient mutant operators,'' *ACM Trans. Softw. Eng. Methodology*, vol. 5, no. 2, pp. 99–118, 1996, doi: 10.1145/227607.227610.

[124] S. Sidiroglou-Douskos, et al., ''Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement,'' in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*., 2015, pp. 473–486, doi: 10.1145/2694344.2694389.

[125] N. Nethercote and J. Seward, ''Valgrind: A framework for heavy-weight dynamic binary instrumentation,'' *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007, doi: 10.1145/1273442.1250746.

[126] J. King, ''Symbolic execution and program testing,'' *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976, doi: 10.1145/360248.360252.

[127] S. Jha, S. Gulwani, S. S. A, and A. Tiwari, ''Oracle-guided component-based program synthesis,'' in *Proc. Int. Conf. Softw. Eng.*, 2010, pp. 215–224, doi: 10.1145/1806799.1806833.

[128] S. Mechtaev, J. Yi, and A. Roychoudhury, ''Directfix: Looking for simple program repairs,'' in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 448–458, doi: 10.1109/ICSE.2015.63.

[129] J. Xuan, et al., ''Nopol: Automatic repair of conditional statement bugs in Java programs,'' *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017, doi: 10.1109/TSE.2016.2560811.

[130] T. Durieux and M. Monperrus, ''Dynamoth: dynamic code synthesis for automatic program repair,'' in *Proc. Int. Workshop Autom. Softw. Test*, 2016, pp. 85–91, doi: 10.1109/AST.2016.021.

[131] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, ''Automated concurrency-bug fixing.'' in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 221–236.

[132] H. Liu, Y. Chen, and S. Lu, ''Understanding and generating high quality patches for concurrency bugs,'' in *Proc. Int. Symp. Found. Softw. Eng.*, 2016, pp. 715–726, doi: 10.1145/2950290.2950309.

[133] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar, ''Test-driven repair of data races in structured parallel programs,'' *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 15–25, 2014, doi: 10.1145/2666356.2594335.

[134] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, ''Efficient data race detection for async-finish parallelism,'' in *Proc. Int. Conf. Runtime Verification*, 2010, pp. 368–383, doi: 10.1007/978-3-642-16612-9_28.

[135] P. Liu and C. Zhang, ''Axis: Automatically fixing atomicity violations through solving control constraints,'' in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 299–309, doi: 10.1109/ICSE.2012.6227184.

[136] P. Liu, O. Tripp, and C. Zhang, ''Grail: context-aware fixing of concurrency bugs,'' in *Proc. Int. Symp. Foundations Softw. Eng.*, 2014, pp. 318–329, doi: 10.1145/2635868.2635881.

[137] Y. Lin and S. Kulkarni, ''Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 237–247, doi: 10.1145/2610384.2610398.

[138] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, ''Automated repair of HTML generation errors in PHP applications using string constraint solving,'' in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 277–287, doi: 10.1109/ICSE.2012.6227186.

[139] M. Alkhalaf, A. Aydin, and T. Bultan, ''Semantic differential repair for input validation and sanitization,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 225–236, doi: 10.1145/2610384.2610401.

[140] F. Yu, M. Alkhalaf, and T. Bultan, ''Patching vulnerabilities with sanitization synthesis,'' in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 251–260, doi: 10.1145/1985793.1985828.

[141] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, ''MintHint: Automated synthesis of repair hints,'' in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 266–276, doi: 10.1145/2568225.2568258.

[142] Y. Pei, C. F. A, M. Nordio, and B. Meyer, ''Automated program repair in an integrated development environment,'' in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 681–684, doi: 10.1109/ICSE.2015.222.

[143] F. Logozzo and T. Ball, ''Modular and verified automatic program repair,'' *ACM SIGPLAN Notices*, vol. 47, no. 10, 2012, pp. 133–146, doi: 10.1145/2398857.2384626.

[144] M. Fähndrich and F. Logozzo, ''Static contract checking with abstract interpretation,'' in *Proc. Int. Conf. Formal Verification Object-Oriented Softw.*, 2010, pp. 10–30, doi: 10.1007/978-3-642-18070-5_2.

[145] F. Gao, L. Wang, and X. Li, ''Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities,'' in *Proc. Int. Conf. Autom. Softw. Eng.*, 2016, pp. 786–791, doi: 10.1145/2970276.2970282.

[146] A. Abadi, R. Ettinger, Y. Feldman, and M. Shomrat, ''Automatically fixing security vulnerabilities in Java code,'' in *Proc. Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2011, pp. 483–502, doi: 10.1109/SP.2017.26.

[147] S. Ma, D. Lo, T. Li, and R. H. Deng, ''CDRep: Automatic repair of cryptographic misuses in Android applications,'' in *Proc. Asia Conf. Comput. and Commun. Secur.*, 2016, pp. 711–722, doi: 10.1145/2897845.2897896.

[148] M. Egele, D. B. Y., Fratantonio, and C. Kruegel, ''An empirical study of cryptographic misuse in Android applications,'' in *Proc. Conf. Comput. Commun. Secur.*, 2013, pp. 73–84, doi: 10.1145/2508859.2516693.

[149] Z. Coker and M. Hafiz, ''Program transformations to fix C integers,'' in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 792–801, doi: 10.1109/ICSE.2013.6606625.

[150] M. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid, ''A case for automated debugging using data structure repair,'' in *Proc. Int. Conf. Autom. Softw. Eng.*, 2009, pp. 620–624, doi: 10.1109/ASE.2009.92.

[151] B. Elkarablieh, I. Garcia, Y. Suen, and S. Khurshid, ''Assertion-based repair of complex data structures,'' in *Proc. Int. Conf. Autom. Softw. Eng.*, 2007, pp. 64–73, doi: 10.1145/1321631.1321643.

[152] S. Khoshnood, M. Kusano, and C. Wang, ''ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 165–176, doi: 10.1145/2771783.2771798.

[153] M. Selakovic and M. Pradel, ''Automatically fixing real-world javascript performance bugs,'' in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 3–4, doi: 10.1145/2048147.2048149.

[154] C. Le Goues, et al., ''The manybugs and introclass benchmarks for automated repair of C programs,'' *IEEE Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015, doi: 10.1109/TSE.2015.2454513.

[155] M. Böhme and A. Roychoudhury, ''CoREBench: Studying complexity of regression errors,'' in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, 2014, pp. 105–115, doi: 10.1145/2610384.2628058.

[156] Lincoln University of Nebraska, ''Software-artifact infrastructure repository,'' (2017). [Online]. Available: http://sir.unl.edu/

[157] R. Just, D. Jalali, and M. D. Ernst, ''Defects4J: A database of existing faults to enable controlled testing studies for Java programs,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440, doi: 10.1145/2610384.2628055.

[158] V. Dallmeier and T. Zimmermann, ''Extraction of bug localization benchmarks from history,'' in *Proc. Int. Conf. Autom. Softw. Eng.*, 2007, pp. 433–436, doi: 10.1145/1321631.1321702.

[159] F. Long and M. Rinard, ''An analysis of the search spaces for generate and validate patch generation systems,'' in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 702–713, doi: 10.1145/2884781.2884872.

[160] C. Le Goues, W. Weimer, and S. Forrest, ''Representations and operators for improving evolutionary software repair,'' in *Proc. Conf. Genetic Evol. Comput.*, 2012, pp. 959–966, doi: 10.1145/2330163.2330296.

[161] X. Kong, L. Zhang, W. E. Wong, and B. Li, ''Experience report: How do techniques, programs, and tests impact automated program repair?'' in *Proc. Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2015, pp. 194–204, doi: 10.1109/ISSRE.2015.7381813.

[162] E. Smith, E. Barr, C. Le Goues, and Y. Brun, ''Is the cure worse than the disease? overfitting in automated program repair,'' in *Proc. Joint Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2015, pp. 532–543, doi: 10.1145/2786805.2786825.

[163] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, ''Designing better fitness functions for automated program repair,'' in *Proc. Conf. Genetic Evol. Comput.*, 2010, pp. 965–972, doi: 10.1145/1830483.1830654.

[164] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, ''Does genetic programming work well on automated program repair?'' in *Proc. Int. Conf. Comput. Inform. Sci.* 2013, pp. 1875–1878, doi: 10.1109/ICCIS.2013.490.

[165] M. Martinez and M. Monperrus, ''Mining repair actions for guiding automated program fixing,'' Arxiv, Tech. Rep. 1311.3414, 2012.

[166] M. Martinez and M. Monperrus, ''Mining software repair models for reasoning on the search space of automated program fixing,'' *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 176–205, 2015, doi: 10.1007/s10664-013-9282-8.
[167] M. Martinez, W. Weimer, and M. Monperrus, ''Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches,'' in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 492–495, doi: 10.1145/2591062.2591114.
[168] S. Sumi, Y. Higo, K. Hotta, and S. Kusumoto, ''Toward improving graftability on automated program repair,'' in *Proc. Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 511–515, doi: 10.1109/ICSM.2015.7332504.
[169] H. Yokoyama, Y. Higo, K. Hotta, T. Ohta, K. Okano, and S. Kusumoto, ''Toward improving ability to repair bugs automatically: a patch candidate location mechanism using code similarity,'' in *Proc. Symp. Appl. Comput.*, 2016, pp. 1364–1370, doi: 10.1145/2851613.2851770.
[170] S. Tan, H. Yoshida, M. P. R, and A. Roychoudhury, ''Anti-patterns in search-based program repair,'' in *Proc. Int. Symp. Foundations Softw. Eng.*, 2016, pp. 727–738, doi: 10.1145/2950290.2950295.
[171] Y. Qi, X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, ''Empirical effectiveness evaluation of spectra-based fault localization on automated program repair,'' in *Proc. Comput. Softw. Appl. Conf.*, 2013, pp. 828–829, do: 10.1109/COMPSAC.2013.139.
[172] Y. Qi, X. Mao, Y. Lei, and C. Wang, ''Using automated program repair for evaluating the effectiveness of fault localization techniques,'' in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 191–201, doi: 10.1145/2483760.2483785.
[173] X. D. Le, T. B. Le, and D. Lo, ''Should fixing these failures be delegated to automated program repair?'' in *Proc. Int. Symp. Softw. Rel. Eng.*, 2015, pp. 427–437, doi: 10.1109/ISSRE.2015.7381836.
[174] J. H. Perkins, et al., ''Automatically patching errors in deployed software,'' in *Proc. Symp. Operating Syst. Principles*, 2009, pp. 87–102, doi: 10.1145/1629575.1629585.
[175] S. Sidiroglou and A. Keromytis, ''Countering network worms through automatic patch generation,'' *IEEE Secur. Privacy*, vol. 3, no. 6, pp. 41–49, 2005, doi: 10.1109/MSP.2005.144.
[176] E. F. Rizzi, S. Elbaum, and M. B. Dwyer, ''On the techniques we create, the tools we build, and their misalignments: A study of KLEE,'' in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 132–143, doi: 10.1145/2884781.2884835.

Luca Gazzola received the MSc degree in computer science from the University of Milano Bicocca. He is currently working toward the PhD degree at the University of Milano Bicocca working under the supervision of Prof. Leonardo Mariani. His PhD work focuses on software testing, and more specifically on field testing. He is also interested in automatic software testing and automatic software repair.

Daniela Micucci received the PhD degree in mathematics, statistics, computational sciences and computer science from the University of Milano, in 2004. She is currently an assistant professor at the University of Milano Bicocca. Her research interests include software engineering, in particular software architectures, real-time systems, and self-healing and self-repairing systems. She is currently active in several European and National projects. She is also regularly involved in the program committees of workshops and conferences in her areas of interest. She is a member of the IEEE.

Leonardo Mariani received the PhD degree in computer science from the University of Milano Bicocca, in 2005. He is an associate professor at the University of Milano Bicocca. His research interests include software engineering, in particular software testing, static and dynamic analysis, automated debugging, and self-healing and self-repairing systems. He has been awarded with the ERC Consolidator Grant in 2015 and he is currently active in several European and National projects. He is regularly involved in the organizing and program committees of major software engineering conferences. He is a senior member of the IEEE.