# Automated Program Repair

Claire Le Goues
Carnegie Mellon University
clegoues@cs.cmu.edu

Michael Pradel
TU Darmstadt
michael@binaervarianz.de

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

## ABSTRACT

Automated program repair can greatly relieve programmers from the burden of manually fixing the ever increasing number of programming mistakes. At the same time, achieving such a goal involves solving technical challenges in scalability, patch quality, and integration into developer workflows. This article presents an overview of the state-of-the-art tools and techniques in automated program repair. We also take a forward looking view of the area by presenting emerging use cases for program repair, such as online programming education and patching of vulnerabilities.

## KEYWORDS

Bugs, Genetic programming, Symbolic Execution, Learning

**ACM Reference Format:**
Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2020. Automated Program Repair. In *Proceedings of Communications of the ACM (CACM).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Alex is a software developer, a recent hire at the company of her dreams. She is finally ready to push a highly anticipated new feature to the shared code repository, an important milestone in her career as a developer. As is increasingly common in development practice, this kind of push triggers myriads of tests that the code needs to pass before becoming visible to everyone in the company. Alex has heavily tested the new feature, and is confident that it will pass all the tests automatically triggered by the push. Unfortunately, Alex learns that build system rejected the commit. The continuous integration system reports failed tests, associated with a software package developed by a different team entirely. Alex now needs to understand the problem and fix the feature *manually*.

What if, instead of simply informing Alex of the failing test, the build system also suggested one or two possible patches for the committed code? Although this general use case is still fictional, a growing community of researchers is working on new techniques for *automated program repair* that could make it a reality. A bibliography of automated program repair research appears in [Mon17].

In essence, automated repair techniques try to automatically identify patches for a given bug[1], which can then be applied with little, or possibly even without, human intervention. This type of work is beginning to see adoption in certain, constrained, practical domains. Static bug finding tools increasingly provide "quick fix" suggestions to help developers address flagged bugs or bad code patterns, and Facebook recently announced a tool that automatically suggests fixes for bugs found via their automatic testing tool for Android applications [MBC+19].

The problem of bugs motivates a broad array of work on automatically identifying them. Advances in formal verification have shown the promise of fully-assured software. However, the pace and scale of modern software development often precludes the application of such techniques from all but the most safety-critical systems. Lighter-weight static approaches that rely most commonly on syntactic pattern matching or less complex static analysis are becoming increasingly popular as quality gates in many companies [SAE+18, JSMHB13]. Testing, at multiple levels of system abstraction, remains the most common bug detection technique in practice.

While detecting bugs is a necessary step toward improving software, it leaves the arguably harder task of *fixing* bugs unsolved. In practice, program repair is challenging for several reasons. A developer must at first understand the problem and localize its root cause in the source code. Next, she must speculate about strategies to possibly fix the problem. For some of these strategies, the developer will evaluate a potential patch, by applying it and evaluating whether the associated test cases then pass; if not, she might use the failing test cases to conduct additional debugging activities. Finally, the developer must select a patch and apply it to code base. The difficulty of all these tasks is compounded by the fact that complex software projects tend to contain legacy code, code written by other members of an organization, or even code written by third-parties.

The promise of automated program repair is in reducing the burden of these tasks by suggesting likely correct patches for software bugs. At a high level, such techniques take as input a program and some specification of the correctness criteria that the fixed program should meet. Most research techniques use test suites for this purpose: one or more failing tests indicate a bug to be fixed, while passing tests indicate behavior that should not change. The end goal is a set of program changes (typically to source code) that leads *all* tests to pass, fixing the bug without breaking other behavior.

The grand challenge in today's research on automated program repair is the problem of weak specifications. Since detailed formal specifications of intended program behavior are typically unavailable, program repair is driven by weak correctness criteria, such as a test-suite. As a result, the generated patches may over-fit the given test-suite, and may not generalize to tests outside the test-suite.

In the rest of the article, we discuss some of the technical developments in automated program repair, including an illustration of the over-fitting problem. We start by sketching some of the use-cases of automated program repair.

[1]We use the colloquial term "bug" to refer to programming mistakes that result in unintended runtime behavior.


*CACM, 2019*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 2 USE CASES

This section discusses four practical use cases of automated repair, and reports initial experience based on current repair techniques.

### 2.1 Fixing Bugs throughout Development

Existing continuous integration (CI) pipelines, such as Jenkins, are an important stepping stone for integrating repair into the development process. By regularly building, testing, and deploying a code base, CI provides the prerequisites for repair tools that use test suites as correctness specifications. Repair can become an activity in CI systems that suggests patches in response to regression test failures, such as for Alex, our hypothetical programmer.

Are we there yet? Existing techniques for automated repair of correctness bugs are typically evaluated for effectiveness using bugs taken from open source projects. Because many techniques require input tests to trigger the bug under repair and to evaluate the technique, such programs and bugs must be associated with one or more failing test cases. These bugs are typically collected systematically by going back in time through code histories to identify bug-fixing commits and the regression tests associated with them. Open source projects whose bugs have been studied in this way include popular Java projects, e.g., various Apache libraries, Log4J, and the Rhino JavaScript interpreter, as well as popular C projects, e.g., the PHP and Python interpreters, the Wireshark network protocol analyzer, and the libtiff library.

Recently, the Repairnator project [UYSM18] has presented a bot which monitors for software errors, and automatically find fixes using repair tools. Another recent work from Facebook [MBC+19] describes experiences in integrating repair as part of continuous integration: a repair tool monitors test failures, reproduces them and automatically looks for patches. Once patches are found they are presented to the developers for validation. Currently the effort focuses on automatically repairing crashes in Android apps, however the project plan is to extend the work to general purpose repair.

### 2.2 Repairing Security Vulnerabilities

Many security vulnerabilities are exploitable memory errors or programming errors, and hence a relevant target for automated repair. Key software including popular libraries processing file formats, or operating system utilities are regularly and rigorously checked for vulnerabilities in response to frequent updates, using grey-box fuzz testing tools, such as American Fuzzy Lop (AFL[2]). Microsoft recently announced the Springfield project; Google similarly announced the OSS-Fuzz project. Such continuous fuzzing work-flows generate use cases for automated program repair. In particular, repair tools can receive tests produced by grey-box fuzz testing tools like AFL.

Are we there yet? Existing repair techniques are effective at fixing certain classes of security vulnerabilities, specifically integer and buffer overflows. An empirical study conducted on OSS Fuzz subjects[3] shows that integer overflow errors are amenable to one-line patches, which are easily produced by repair tools. For example, these changes add explicit casts of variables or constants, modify conditional checks to prevent overflows, or change type declarations. Existing repair tools [MYR16] have also been shown to automatically produce patches for the infamous Heartbleed vulnerability:

```
if (hbtype == TLS1_HB_REQUEST
   /* the following check being added is the fix */
    && payload + 18 < s->s3->rrec.length) {
  ...
  memcpy(bp, pl, payload);
  ...
}
```

It is functionally equivalent to the developer-provided patch:

```
/* the following check being added is the fix */
if (1 + 2 + payload + 16 > s->s3->rrec.length) return 0;
...
if (hbtype == TLS1_HB_REQUEST) {
  ...
}
```

### 2.3 Intelligent Tutoring

The computer programming learning community is growing rapidly. This growth has increasingly led to large groups of potential learners, with often inadequate teaching support. Automated repair can serve as a key component of intelligent tutoring systems that provide hints to learners for solving programming assignments and that automate the grading of students' programming assignments by comparing them with a model solution.

Are we there yet? While repair-based intelligent tutoring remains an open challenge for now, initial evidence on using program repair like processes for providing feedback to students [SGSL13] or for automatic grading of student assignments [YAK+17] have been obtained. Automated assignment grading can benefit from computation of the "semantic distance" between a student's buggy solution and an instructor's model solution. An important challenge for the future is that programming education requires nuanced changes to today's program repair work-flow, since teaching is primarily focused on guiding the students to a solution, rather than repairing their broken solution.

### 2.4 Self-Healing of Performance Bottlenecks

With the emergence of a wide variety of Internet-of-things (IoT) software for smart devices, drones, and other cyber-physical or autonomous systems, there is an increasing need for *online* program repair, especially for non-functional properties like energy consumption. Consider a drone used for disaster recovery, such as flood or fire control. The drone software may encounter unexpected or perilous inputs simply by virtue of being faced with an unforeseen physical environment, which may drain the device's battery. There exists a need for online self-healing of the drone software. Automated repair targeted at non-functional issues, such as performance bottlenecks, can provide such self-healing abilities.

Are we there yet? Current repair techniques for non-functional properties have shown their effectiveness in improving real-world software. Consider two examples of performance-related repair tools. First, the MemoizeIt tool [TPG15] suggests code that performs application-level caching, which allows programs to avoid unnecessarily repeated computations. Second, the Caramel tool [NCRL15]

[2] http://lcamtuf.coredump.cx/afl/
[3] https://github.com/google/oss-fuzz

```
1 int triangle(int a, int b, int c){
2     if (a <= 0 || b <= 0 || c <= 0)
3           return INVALID;
4     if (a == b && b == c)
5           return EQUILATERAL;
6     if (a == b || b != c)   // bug!
7           return ISOSCELES;
8     return SCALENE;
9 }
```

**Figure 1: Simple example for categorizing triangles.**

| Test-id | a | b | c | Expected output | Pass/Fail |
|---|---|---|---|---|---|
| 1 | -1 | -1 | -1 | INVALID | Pass |
| 2 | 1 | 1 | 1 | EQUILATERAL | Pass |
| 3 | 2 | 2 | 3 | ISOSCELES | Pass |
| 4 | 3 | 2 | 2 | ISOSCELES | Fail |
| 5 | 2 | 3 | 2 | ISOSCELES | Fail |
| 6 | 2 | 3 | 4 | SCALENE | Fail |

**Figure 2: Test suite for the function in Figure 1.**

has suggested patches for a total of 150 previously unknown performance issues in widely used Java and C/C++ programs, such as Lucene, Chromium, and MySQL, that are now fixed based on the suggested repairs. While these examples are encouraging, the question of how to apply non-functional repair for fully automated self-healing remains open.

## 3 SIMPLE EXAMPLE

We now describe a simple example that we will use to illustrate the various state-of-the-art techniques in program repair. The example is selected for didactic purposes rather than to illustrate all the capabilities of repair techniques. Today's techniques apply to significantly more complex programs, as we describe in Section 2.

Consider a function that categorizes a given triangle as scalene, isosceles, or equilateral (Figure 1). From the definition of isosceles triangles learned in middle school, we can see that the condition in line 6 should be rectified to

```
(a == b || b == c || a == c)
```

This modification is non-trivial; it goes beyond simply mutating one operator in the condition.

The test suite in Figure 2 captures the various triangle categories considered by the function: INVALID, EQUILATERAL, ISOSCELES and SCALENE. Because the code contains a bug, several of the tests fail. The goal of automated program repair is to take a buggy program and a test suite, such as these, and produce a patch that fixes the program. The test suite provides the correctness criterion in this case, guiding the repair toward a valid patch. In general, there may exist any number of patches for any particular bug, and even humans can find different patches for real-world bugs.

At a high level, the program repair problem can be seen as follows: *Given a buggy program P, and a test suite T such that P fails one or more tests in T, find a "small" modification of P such that the modified program P' passes T.* The term "small" simply refers to the fact that developers generally prefer a simpler patch over a complicated one. Some techniques even try to find a minimal patch. Others tradeoff patch size with other goals, such as finding a patch efficiently. A particular risk in automated repair is a "patch" that causes the provided test cases to pass but that does not generalize to the complete, typically unavailable, specification. That is, the patch produced by an automated repair method can *overfit* the test data. An extreme case of an overfitted repair for the tests in Figure 2 is the following:

```
if (a==-1 && b==-1 && c==-1)
  return INVALID;
if (a==1 && b==1 && c==1)
  return EQUILATERAL;
if (a==2 && b==2 && c==3)
  return ISOSCELES;
...
```

Of course, such a "repaired" program is not useful since it does not pass any tests outside the provided test suite. This example is deliberately extreme. More commonly, patches produced by current repair techniques tend to overfit the provided test suite by disabling (or deleting) under-tested functionality [SBGB15].

## 4 STATE-OF-THE-ART

Automatically repairing a bug involves (implicitly) searching over a space of changes to the input source code. Techniques for constructing such patches can be divided into broad categories, based on what types of patches are constructed, and how the search is conducted. Figure 3 gives an overview of the techniques. The inputs to these techniques are a buggy program and a correctness criterion (the correctness criterion is often given as a test-suite). Most techniques start with a common preprocessing step that identifies those code locations that are likely to be buggy. Such a *fault localization* procedure, e.g., [JHS02], provides a ranking of code locations that indicates their potential buggy-ness. At a high level, there are two main approaches, *heuristic repair* and *constraint-based repair*. These techniques can sometimes enhanced by machine learning, which we call *learning-aided repair*.

### 4.1 Heuristic Repair

Heuristic search methods, shown at the left of Figure 3 employ a generate-and-test methodology, constructing and iterating over a search space of syntactic program modifications. These techniques can be explained schematically as follows.

*for* *cand* ∈ *SearchSpace do*
  *validate cand // break if successful*
*done*

with *SearchSpace* denoting the set of considered modifications of the program. Validation involves calculating the number of tests that pass when a suggested patch has been applied. This can amount to a fitness function evaluation in genetic programming or other stochastic search methods.

Heuristic repair operates by generating patches that transform the program Abstract Syntax Tree (AST). An AST is a tree-based representation of program source code that captures important program constructs, while abstracting away syntactic details like parentheses or semicolons. Given fault localization information that

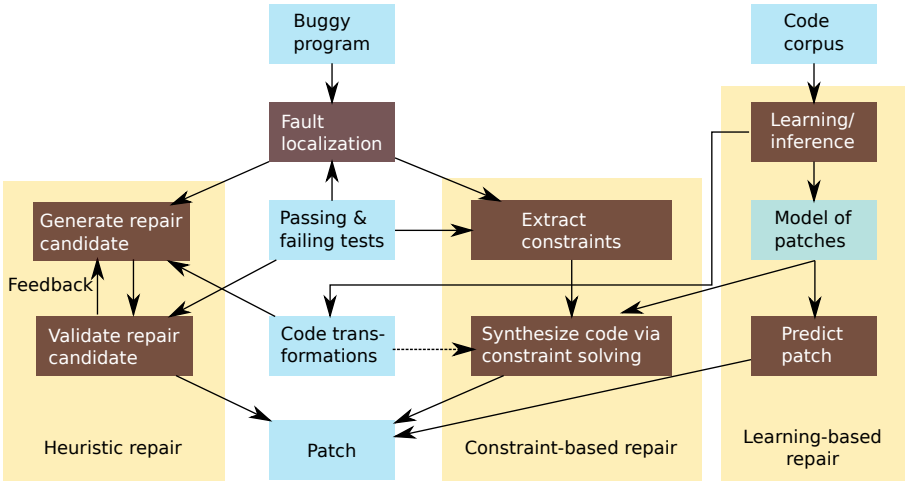


**Figure 3: Overview of repair techniques.**

pinpoints code locations in the program that are the most likely to be buggy, syntactic techniques render the search tractable by making choices along one of three axes which are described next: mutation selection, test execution, and the traversal strategy.

*Mutation selection.* Due to the combinatorial explosion of possible mutations, the number of program variants that can be generated and compiled is typically very large. Techniques thus must limit the type and variety of edits considered for a repair candidate. This in turn defines the search space, with which search-based repair algorithms have great flexibility. However, this flexibility comes at a risk: If the search space is too small, the desired repair may not even be in the search space. For our triangle example (Figure 1), recall that the most natural patch replaces line 6 with `(a == b || b == c || a == c)`. If we only consider mutations that modify binary operators, the single-edit search space of the repair algorithm will not contain the developer-provided repair, which requires augmenting the branch condition with new conditions. On the other hand, if the search space is too large, the search can become intractable, such that the repair may not be found by the algorithm in a reasonable amount of time.

To address this issue, some techniques limit edits to only deletion, insertion, or replacement of code at the statement- or block-level. For code insertion or replacement, a common approach is to pull code from elsewhere in the same program or module, following the *plastic surgery* hypothesis (that correct code can be imported from elsewhere in the same program) [Har10] or the *competent programmer hypothesis* (that programmers are mostly competent, and while they may make a mistake in one portion of a program, they are likely to have programmed similar logic correctly, elsewhere). Such a technique would therefore only consider moving entire blocks or lines of code around, e.g., an entire `if` condition semantically similar to the one shown in Figure 1. This can often work by virtue of the fact that source code is repetitive [RHG+16].

Other techniques have benefited from using more expressive transformation templates, such as guiding a de-reference operation with a null-pointer check. Such transformation templates trade off repair space size for readability and "naturalness" of the resulting patches. Moving from statement-level edits to expression-level edits increases the search space, with the amount of increase depending on the transformation templates used to construct the search space.

However, of course, even if the search space is large, the mutation operators may not support the behavioral change needed by the program, or may affect the desired change in ways different from what a human might propose. A technique that may modify operators or insert conditions (copied from elsewhere in the program) would still struggle on this small program, since `(a == c)` never appears verbatim in our example. Such a lack of correct code fragments can result in degenerate behavior on smaller programs that provide little repair material. It also motivates research in intelligently augmenting the search space, e.g., by considering past versions of a program.

*Test execution.* Repair candidates are evaluated by running the modified program on the provided set of test cases. Test execution is typically the most expensive step, as test suites can be large and techniques may need to rerun them many times. Various strategies have been proposed to reduce this cost, including test suite selection and prioritization. Search strategies that do not require a fitness function to guide evolution, e.g., based on random or deterministic search, can reduce the cost of testing by simply failing early (at the first test failure). Moreover, such techniques may run the test cases in a heuristic order designed to maximize the chance that, if a test case is going to fail, it is run early in the validation process.

*Traversal strategy.* Finally, techniques vary in how they choose which patches to explore, and in what order. GenProg [WNGF09], an early technique proposed in this domain, uses a genetic programming heuristic that evolves program patches towards a solution. This approach uses a fitness function based on the number of test cases passed by a patched program. Subsequent techniques like PAR [KNSK13] have followed, varying in the mutation operators (PAR) or the fitness function. Other techniques simply sample randomly typically restricting themselves to single-edit patches

[QML+14], or in a heuristic, deterministic order as in GenProg AE [WFF13].

## 4.2 Constraint-based Repair

In contrast to heuristic repair techniques, constraint-based techniques proceed by constructing a *repair constraint* that the patched code should satisfy, as illustrated in Figure 3. The patch code to be generated is treated as a function to be synthesized. Symbolic execution or other approaches extract properties about the function to be synthesized; these properties constitute the repair constraint. Solutions to the repair constraint can be obtained by constraint solving or other search techniques. In these approaches, formulation of the repair constraint is the key, not the mechanism for solving it. This class of techniques can be captured via the following schematic:

| | |
|---|---|
| *for* | *test* $t \in$ *test-suite* $T$ |
| | *compute* repair constraint $\psi_t$ |
| *synthesize* | $e$ as solution for $\bigwedge_t \psi_t$ |

In this case, $T$ is the test-suite used as the correctness criterion to guide repair. The constraint $\psi_t$ will be computed via a symbolic execution [Kin76] of the path traversed by test $t \in T$. The constraint $\psi_t$ is often of the form

$$\psi_t \equiv \pi_t \wedge output = expected$$

where $\pi_t$ is the path condition of the path traversed by test $t$, *output* is the symbolic expression capturing the output variable in the execution of $t$ and *expected* captures the oracle or expectation. The path condition of a program path is a formula which is true for those inputs which traverse the path [Kin76].

*Computing repair constraints and angelix values.* To illustrate constraint-based repair, reconsider our running example from Section 3. SemFix [NQRC13], which is representative for constraint-based techniques, substitutes the faulty condition in line 6 with an abstract function $f(a, b, c)$ on the live variables. In this example, $f$ is a predicate that takes the integer values $a, b, c$ and returns true/false. Then, the technique symbolically executes the tests in the given test-suite to infer properties about the unknown function $f$. These properties are crucial for synthesizing an appropriate $f$ that can pass all the given test cases.

The first two tests in our test suite do not even reach line 6. Hence, SemFix will not infer any property about the function $f$ from them. From the last four tests, it can infer the repair constraint

$$f(2, 2, 3) \wedge f(3, 2, 2) \wedge f(2, 3, 2) \wedge \neg f(2, 3, 4)$$

This is because analysis of the program has revealed that for input exercising line 6 if $f$ is true, the program returns `ISOSCELES` and otherwise `SCALENE`.

Inferring detailed constraint specifications can be difficult, sometimes posing significant scalability issues. This motivates more efficient inference of value-based specifications [MYR16]. In particular, *angelic values* are inferred for patch locations, where an angelic value for a fix location is one that makes a failing test case pass. Once (collections of) angelic values are identified for each failing test, program synthesis can then be employed to generate patch code meeting such a value-based specification. This is the philosophy embodied in the Angelix tool [MYR16] where angelic values are obtained via symbolic execution (instead of producing repair specifications in the form of SMT constraints via symbolic execution directly). This way of dividing the repair task into (a) angelic value determination, and (b) patch code generation to meet angelic values, is symptomatic of semantic repair approaches.

Instead of obtaining angelic values by symbolic execution and constraint solving, they may also be obtained by search, particularly for conditional statements. This is because each occurrence of a conditional statement has only two possible return values: true and false. Techniques that work on enumerating possible angelic values without adopting symbolic execution [XMD+17, LR15] typically try to repair conditional statements exclusively, where the angelic values are exhaustively enumerated until all failing test cases pass. Such techniques adopt the work-flow of semantic repair techniques (specification inference followed by patch generation), with an enumeration step fully or partially replacing symbolic program analysis. Symbolic analysis based approaches such as [MYR16] on the other hand, avoid exhaustive enumeration of possible angelic values.

*Solving constraints to find a patch.* Once repair constraints or angelic value(s) of a statement to be fixed are obtained, these techniques need to generate a patch to realize the angelic value. Finding a solution to the repair constraint yields a definition of the abstract function $f$, which corresponds to the patched code. This is often achieved by either search or constraint solving, where the operators allowed to appear in the yet-to-be synthesized function $f$ are restricted. In the above example, if we restrict the operators allowed to appear in $f$ to be relational operators most search or solving techniques will find the expression `a == b || b == c || a == c`. Efficient program synthesis techniques (see [ASFSL18] for an exposition of some recent advances in program synthesis) are often used to construct the function $f$.

## 4.3 Learning-based Repair

Recent improvements in advanced machine learning, especially deep learning, and the availability of large numbers of patches enable learning-based repair. Current approaches roughly fall into three categories, which vary by the extent to which they exploit learning during the repair process. One line of work [LR16] learns from a corpus of code a model of correct code, which indicates how likely a given piece of code is w.r.t. the code corpus. The approach then uses this model to rank a set of candidates patches, to suggest the most realistic patches first. Another line of work infers code transformation templates from successful patches in commit histories [LAR17, BVLR17]. In particular, [LAR17] infers AST-to-AST transformation templates that summarize how patches modify buggy code into correct code These transformation templates can then be used to generate repair candidates.

The third line of work not only improves some part of the repair process through learning, but trains models for end-to-end repair. Such a model predicts for a given piece of buggy code the repaired code, without relying on any other explicitly provided information. In particular, in contrast to the repair techniques in earlier sections, such models do not rely on a test suite or a constraint solver. DeepFix [GPKS17] trains a neural network that fixes compilation errors, e.g., missing closing braces, incompatible operators,

or missing declarations. The approach uses a compiler as an oracle to validate patch candidates before suggesting them to the user. Tufano et al. [TWB+18] propose a model that predicts arbitrary fixes and train this model with bug fixes extracted from version histories. According to their initial results, the model produces bug-fixing patches for real defects in 9% of the cases. Both approaches abstract the code before feeding it into the neural network. For the running example in Figure 1, this abstraction would replace the application-specific identifiers `triangle` and `EQUILATERAL` with generic placeholders, such as `VAR1` and `VAR2`. After this abstraction, both approaches use an RNN-based sequence-to-sequence network that predicts how to modify the abstracted code.

Given the increasing in interest in learning-based approaches toward software engineering problems, we will likely see more progress on learning-based repair in the coming years. Key challenges toward effective solutions include finding an appropriate representation of source code changes and obtaining large amounts of high-quality human patches as training data.

### 4.4 Repair of Non-Functional Properties

To help developers improve the software efficiency, several approaches identify optimization opportunities and make suggestions on how to refactor the code to improve performance. These approaches typically focus on a particular kind of performance problem, e.g., unnecessary loop executions [NCRL15] or repeated executions of the same computation [TPG15]. Another line of work selects which data structure, out of a given set of functionally equivalent data structures, is most likely to provide the best performance for a given program [SVY09]. All these approaches suggest code changes but leave the final decision whether to apply an optimization to the developer.

To mitigate security threats, various techniques for repairing programs at runtime have been proposed. These approaches automatically rewrite code to add a runtime mechanism that enforces some kind of security policy. For example, such repair techniques can enforce control flow integrity [ABEL05], prevent code injections [SW06], automatically insert sanitizers of untrusted input, or enforce automatically inferred safety properties [PKL+09].

We note that existing techniques to repair non-functional properties typically focus on a particular kind of problem, e.g., a kind of performance anti-pattern or attack. This distinguishes them from the core repair literature for fixing correctness bugs, which typically aim at fixing a larger set of errors.

## 5 PERSPECTIVES AND CHALLENGES

Despite tremendous advances in program repair during the last decade, there remain various open challenges to be tackled by future work. We identify three core challenges: increasing and ensuring the *quality* of repairs, extending the *scope* of problems addressed by repair, and integrating repair into the *development process*.

### 5.1 Quality

The quality challenge is about increasing the chance that an automatically identified repair provides a *correct* fix that is *easy to maintain* in the long term. Addressing this challenge is perhaps the most important step toward real-life adoption of program repair.

*Measures of Correctness.* An important aspect of fix quality is whether the fix actually corrects the bug. In practice, program repair relies on measures of correctness. Finding such a measure is a difficult and unsolved problem, which applies both to patches produced by humans and by machines. To date, researchers have assessed quality using human judgment, crowdsourced evaluations, comparison to developer patches of historical bugs, or patched program performance on indicative program workloads or held-out test cases. The recent work of [XLZ+18] provides a novel outlook for filtering patches based on the behavior of the patched program vis-a-vis the original program on passing and failing tests.

*Alternative Oracles.* The bulk of the existing literature focuses on test-based repair where the correctness criteria is given as a test suite. Richer correctness properties, e.g., assertions or contracts, can be used to guide repair, when available [WPF+10]. Other approaches consider alternative oracles, such as potential invariants inferred from dynamic executions [PKL+09]. Such approaches can follow the "bugs as deviant behavior" philosophy, where deviations of an execution from "normal" executions are observed and avoided. In particular, [WFK+16] provides an overview of various (partial) oracles that can be used for repair.

*Correctness Guarantees.* Few of today's repair techniques provide any guarantees about the correctness of produced patches, which can hinder the application of automated repair, especially to safety-critical software. If correctness guarantees needed are available as properties, such as pre-conditions, post-conditions and object invariants, these can be used to guide program repair. The work of [LB12] reports such an effort where repair attempts to increase the number of property preserving executions, while reducing the number of violating executions. However, such formal techniques are contingent on the properties to drive the repair being available.

*Maintainability.* Once a correct fix has been detected and applied to the code base, the fixed code should be as easy to maintain as a human fix. Initial work in this domain has investigated the effect that automatically-generated patches impact human maintenance behavior [FLW12]. More study is needed to develop a foundational understanding of change quality, especially with respect to the human developers who will interact with a modified system.

A promising avenue for tackling the quality challenge is by leveraging information available from other development artifacts, including documentation or formal specifications, language specifications and type systems, or source control histories of either the program under repair or of the broad corpora of freely available open-source software. Such additional information can reduce the repair search space by imposing new constraints on potential program modifications (e.g., as suggested by a type system) and increase the probability that the produced patch is human-acceptable.

### 5.2 Scope

The scope challenge is about further extending the kinds of bugs and programs to which automated repair applies.

*General-purpose repair.* Research in program analysis has long focused on special-purpose repair tools for specific kind of errors, such as buffer overflow errors [PKL+09], or bugs in domain-specific

languages [SSA+12]. More recent work, discussed in Section 4, focuses on general-purpose repair tools, which do not make any assumptions about the kind of bugs under consideration. While automatically fixing all bugs seems out of reach in the foreseeable future, targeting a broad set of bugs remains on important challenge.

*Complex programs and patches.* Many of the key innovations in the initial research in program repair concerned the scalability of techniques to complex programs. For example, search-based techniques moved from reasoning over populations of program ASTs to populations of small edit programs (the patches themselves) and developed other techniques to effectively constrain the search space. Constraint-based repair strategies have moved from reasoning about the semantics of entire methods to only reasoning about the desired change in behavior. These efforts enable scaling to programs of significant size, and multi-line repairs [MYR16]. We anticipate that scalability will periodically return to the fore as program repair techniques engage in more complex reasoning. We emphasize here that program repair techniques should remain scalable with respect to large programs as well as large search spaces (complex changes).

### 5.3 Development Process

The final challenge is about integrating repair tools into the development process.

*Integration with bug detection.* Bug detection is the natural step preceding program repair. It is possible to fuse debugging and repair into one step, by viewing repair as the minimal change which makes the program pass the given tests. We envision future work integrating repair with bug detection techniques, such as static analysis tools. Doing so may enable repair techniques to obtain additional information about possible repairs from static analyses, in addition to the test cases used nowadays. As a first step in this direction, a static analysis infrastructure used at Google suggests fixes for a subset of its warnings [SAE+18]. A promising future direction here could be to extend static analysis tools for generating dynamic witnesses or scenarios of undesirable behavior.

*IDE integration.* Most of today's repair tools are research prototypes. Bringing these tools to the fingertips of developers in a user-friendly fashion will require efforts toward integrating repair into Integrated Development Environments (IDEs). For example, an IDE-integrated repair tool could respond to either failed unit or system tests or developer prompting. To the best of our knowledge, this application has not yet been widely explored. Suitably interactive response times are a precondition for such an approach. This research direction will benefit from interaction with experts in developer tooling and human-computer interaction, to ensure that tools are designed and evaluated effectively.

*Interactivity.* As program repair gets integrated into development environments, interacting with the developer *during* repair is important. While the focus in the past decade has been on fully automated repair, putting the developer back into the loop is necessary, in particular, due to the weak specifications (test suites) often used to guide program repair. User interactivity may be needed to yield expected outputs of additional test inputs which are generated to strengthen the test-suite driving repair [SES17]. It is of course possible to filter plausible but possibly incorrect patches by, e.g., favoring smaller patches or favoring patches "similar" to human patches. Nevertheless, the developer still needs to explore the remaining, large set of patch suggestions.

*Explaining repairs.* A strongly related problem is to *explain* repair suggestions. One idea worth pursuing is to compute and present the correlation of patches based on program dependencies and other semantic features, which allow the developer to loosely group together plausible patches. Explaining repairs is needed particularly in its application to programming education [YAK+17]. Instead of merely fixing a students' incorrect program to the model correct program, it is useful for the repair tool to generate hints of what is missing in the students' repair. Such hints may take the form of logical formulae capturing the effect of repairs, which are gleaned from constraint based repair tools; these hints may be presented in natural language, instead of logic, for easy comprehension by the learners.

## 6 CONCLUDING REMARKS

Automated program repair remains an enticing yet achievable possibility, which can improve program quality while improving programmers' development experience.

Technically speaking, automated repair involves challenges in defining, navigating and prioritizing the space of patches. The field thus benefits from past lessons learned in search space definition and navigation in software testing, as embodied by the vast literature in test selection and prioritization. The GenProg tool [WNGF09] is a fitting example of how genetic search, which has been useful for testing, can be successfully adapted for repair. At the same time, automated repair comes with new challenges, because it may generate patches that overfit the given tests. This is a manifestation of tests being incomplete correctness specifications. Thus there is a need for inferring specifications to guide repair, possibly by program analysis. The Semfix and Angelix tools [NQRC13, MYR16] are fitting examples of how the repair problem can be envisioned as one of inferring a repair constraint, and they have shown the scalability of such constraint-based techniques.

Conceptually speaking, automated program repair closes the gap between the huge manual effort spent today in writing correct programs, and the ultimate dream of generating programs automatically via learning approaches. Given the challenges of generating multi-line program fixes in program repair, we can thus imagine the difficulty of generating *explainable* programs automatically.

Pragmatically speaking, automated program repair also makes us keenly aware of the challenges in managing changes in software engineering projects, and the need for automation in this arena. Today, manual debugging and maintenance often takes up 80% of the resources in a software project, prompting practitioners to long declare a legacy crisis [SPL03]. In future program repair can provide tool support by repairing bugs from complex changes in software projects. This can help resolve a dilemma of developers when managing program changes: "Our dilemma is that we hate change and love it at the same time; what we really want is for things to remain the same but get better" (quote by Sydney Harris).

*Acknowledgments.* The authors acknowledge many discussions with researchers at the Dagstuhl seminar 17022 on Automated Program Repair (January 2017). Claire Le Goues acknowledges support of the U.S. National Science Foundation under grants no. CCF-1750116 and CCF-1738253. Michael Pradel acknowledges support of BMWF/Hessen within CRISP and support of the DFG within the ConcSys and Perf4JS projects. Abhik Roychoudhury acknowledges support of National Research Foundation Singapore, National Cybersecurity R&D program (Award No. NRF2014NCR-NCR001-21).

## REFERENCES

[ABEL05] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS)*, pages 340–353. ACM, 2005.

[ASFSL18] R Alur, R Singh, D Fisman, and A Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61:84–93, 2018.

[BVLR17] D.B. Brown, M. Vaughn, B. Liblit, and T.W. Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 511–522, 2017.

[FLW12] Z.P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[GPKS17] R. Gupta, S. Pal, A. Kanade, and S. Shevade. DeepFix: Fixing common C language errors by deep learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2017.

[Har10] M. Harman. Automated patching techniques: the fix is in: technical perspective. *Communications of the ACM*, 53:108–108, 2010.

[JHS02] J.A. Jones, M.J. Harrold, and J. Stasko. Visulization of test information to assist fault localization. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2002.

[JSMHB13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[Kin76] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19, 1976.

[KNSK13] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2013.

[LAR17] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2017.

[LB12] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.

[LR15] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[LR16] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *ACM International Symposium on Principles of Programming Languages (POPL)*, 2016.

[MBC+19] A Marginean, J Bader, S Chandra, M Harman, Y Jia, K Mao, A Mols, and A Scott. Sapfix: Automated end-to-end repair at scale. In *International Conference on Software Engineering (ICSE), Software Enginering in Practice (SEIP) track*, 2019.

[Mon17] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys*, 51(1), 2017.

[MYR16] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2016.

[NCRL15] A. Nistor, P-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

[NQRC13] H.D.T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program Repair via Semantic Analysis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2013.

[PKL+09] J.H. Perkins, S. Kim, S. Larsen, S.P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W-F. Wong, Y. Zibin, M.D. Ernst, and M.C. Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2009.

[QML+14] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *International Conference on Software Engineering (ICSE)*, 2014.

[RHG+16] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 428–439, 2016.

[SAE+18] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, 2018.

[SBGB15] E.K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *International Symposium on Foundations of Software Engineering (FSE)*, 2015.

[SES17] D. Shriver, S. Elbaum, and K.T. Stolee. At the end of synthesis: narrowing program candidates. In *International Conference on Software Engineering (ICSE)*, 2017.

[SGSL13] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[SPL03] R. Seacord, D. Plakosh, and G. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes and Business Practices*. Addison Wesley, 2003.

[SSA+12] 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 *34th International Conference on Software Engineering (ICSE)*, 2012.

[SVY09] O. Shacham, M.T. Vechev, and E. Yahav. Chameleon: adaptive selection of collections. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418. ACM, 2009.

[SW06] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Symposium on Principles of Programming Languages (POPL)*, pages 372–382, 2006.

[TPG15] L.D. Toffola, M. Pradel, and T.R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[TWB+18] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *International Conference on Automated Software Engineering (ASE)*, 2018.

[UYSM18] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. How to design a program repair bot? insights from the repairnator project. In *International Conference on Software Engineering (ICSE), Track Software Engineering in Practice*, 2018.

[WFF13] W. Weimer, Z. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2013.

[WFK+16] W. Weimer, S. Forrest, M. Kim, C. Le Goues, and P. Hurley. Trusted software repair for system resiliency. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) Workshops*, 2016.

[WNGF09] W. Weimer, T.V. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2009.

[WPF+10] Y Wei, Y Pei, C.A. Furia, L.S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[XLZ+18] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. In *International Conference on Software Engineering (ICSE)*, 2018.

[XMD+17] J. Xuan, M. Martinez, F. Demarco, M. Clement, S.L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43, 2017.

[YAK+17] J. Yi, U.Z. Ahmed, A. Karkare, S.H. Tan, and A. Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2017.