# Automatic Program Repair

**Claire Le Goues**, Carnegie Mellon University

**Michael Pradel**, University of Stuttgart

**Abhik Roychoudhury**, National University of Singapore

**Satish Chandra**, Facebook


*Digital Object Identifier 10.1109/MS.2021.3072577*
*Date of current version: 18 June 2021*

**PROGRAMMING MISTAKES OF** all kinds—in source code, configurations, tests, or other artifacts—are a wide-ranging and expensive problem. Developers dedicate a significant proportion of engineering time and effort to finding and fixing bugs in their code, businesses lose market share when vulnerabilities in the software they sell impact customers, and overall productivity is impacted by software that does not work as intended or is prone to vulnerabilities.[1] Rapidly finding and fixing bugs and vulnerabilities only grows in importance as software is continuously evolving and deployed and as society becomes increasingly dependent on software systems in all aspects of modern life. Speaking to this general problem, this special issue of *IEEE Software* addresses recent advances in research and practice in automatic software repair.

## Prologue

Automated program repair[2] is an emerging suite of technologies for automatically fixing errors or vulnerabilities—bugs, colloquially—in software systems. Automatic program repair as a research field focuses on a class of techniques that produces source code-level patches for such bugs, of the same variety that programmers produce in addressing a defect they find in their own programs or in response to a bug report. Thus, at a high level, an automatic repair approach takes as input a program and some evidence that the program has a bug (commonly, a failing test) and produces a patch for that program's source to fix that bug, ideally without negatively influencing other correct functionality.

Overall, automated tooling, analysis, and bots that automatically patch programs are a growing phenomenon in both research and practice. The research community has dedicated increasing attention to this problem, especially over the past decade. This has resulted in a diversity of techniques that seek to fix bugs as identified by failing tests or program crashes, address statically detected violations from automatic bug-finding tools, or repair compilation errors. No matter which bug-identifying method is used, the goal is to find a patch that changes the program appropriately, e.g., by making the tests pass or the crashes go away or enabling the static analyzer pass without complaint. Under the hood, such techniques use technologies ranging from symbolic execution and program synthesis, to machine learning, to evolutionary computation, or increasingly some combination thereof. In engineering practice, recently developed repair tools range from simple "quick-fix" suggestions to address linter checks performed at commit time and vulnerability-suggesting bots on GitHub to, recently at Facebook, more robust automated patching run in production against automatically generated tests.[3]

The goal of this special issue is to provide a checkpoint on the state of research and practice in automatic program repair. The three articles presented are summarized in "New Repair Architectures, Techniques, and Practices."

In this introduction, we introduce key ideas underlying the general field of automatic program repair and provide a brief overview of the structure of the problem and various solutions to provide context for the subsequent material.

## What Is a Bug?

Practically, a significant percentage of a software project's cost today is not spent in the creative activity of software construction but rather in the corrective activity of debugging and fixing errors. However, the task of debugging is inherently complicated. Most systems lack formal specifications describing intended program behavior. Without a formal or systematic documentation of correct behavior, the definition of an "error" or "bug" often resides in the programmer's mind or in the user's sometimes nebulous expectations of program behavior. This can pose a challenge for automatic bug-correction techniques, which require a tangible mechanism to identify the "fault" to be repaired.

Automated techniques for bug detection, mitigation, or prevention have a long history in computer science research. Programming languages and their type systems and compilers can warn programmers when they make certain kinds of mistakes or eliminate them entirely by design. Static analyses, sometimes built into integrated development environments or run at commit time, can flag problematic patterns or even, increasingly, find deep semantic errors. Dynamic self-healing techniques can enforce security or other correctness policies by enforcing control flow integrity, preventing code injection, or automatically sanitizing inputs (see, e.g., the works by Perkins et al.[4] and Serebryany et al.[5]). Such techniques can therefore catch and recover from errors at runtime, without either user or developer intervention.

By contrast, the techniques for automatic software repair we address in this special issue generally aim to produce changes (patches) to the program source code to address the bug altogether (rather than find errors, help programmers avoid errors, or help systems dynamically recover

from them). Sometimes these goals can go hand in hand. For example, some static bug-finding tools increasingly provide the developers with pointers or suggestions to help them understand and fix the underlying problem; indeed, more quick-fix suggestions by bug-finding tools can lead to greater adoption.[6] Similarly, linters or compilers increasingly make suggestions to address flagged errors, and research techniques are being proposed to address more semantically complex bugs, as flagged by static techniques. Such approaches thus use a static bug-finding approach to find a flaw and then can use the static technique to automatically localize the bug and validate that a proposed patch addresses it (i.e., by determining that the static analyzer no longer flags the defect in question).

However, a larger preponderance of current techniques for automatic program repair are dynamic in nature. That is, these methods use failed tests or demonstrated program crashes to demonstrate the existence of a glitch; the goal of the bug-repair process is to modify the program source code so that the test(s) now pass or the program no longer crashes. Other existing program tests are typically used to help the program-repair process avoid unwittingly breaking other desirable behavior, in the same way that continuous integration (CI) test suites help human programmers avoid doing the same in manually modifying their systems. Indeed, some proposed and currently deployed techniques are targeted at that use case exactly: repairing a program with respect to a failed CI test.

## How and When Is It Fixed?

Automatic bug repair is thus, fundamentally, a search problem: the search goal is a set of changes to a program that addresses a given error without introducing new bugs or affecting previously correct behavior. Typically, such techniques begin with automated analysis to localize the bug in question to a smaller set of candidate program locations, one or more of which may correspond to a suitable repair site. This is one of several ways in which automated repair differs, conceptually, from manual bug fixing. In our understanding of manual bug fixing, the programmer first locates the "error" by implicitly considering the intended program behavior and then changes the error location and/or relevant locations to fix the errors there. In automated repair, knowing exactly where the error resides is, strictly speaking, not always necessary. Even if the precise error location—per human judgment—is not accurately

## NEW REPAIR ARCHITECTURES, TECHNIQUES, AND PRACTICES

### NEW REPAIR TOOL ARCHITECTURE

Baudry et al. present an automated repair bot that learns patch patterns from failed builds across projects in their article "A Software-Repair Robot Based on Continual Learning." In terms of advances, this article presents a tool architecture to learn patch patterns; this architecture can potentially be assimilated into continuous integration systems of the future. In terms of challenges, this method repairs an extremely small percentage of the failed builds, meaning that the effectiveness of such an approach remains a work in progress.

### NEW REPAIR TECHNIQUE

The article "A Novel Approach for Search-Based Program Repair" by Trujillo et al. takes a fresh look at the way search is guided in genetic-programming-style program repair, often via an objective function representing the number of tests passed. This article posits that, for automated repair, navigating a diverse set of patches is as important as optimizing an objective function. Toward this goal, the article proposes the use of novelty search to avoid local optima in the search spaces.

### NEW REPAIR PRACTICES

Finally, the article "On the Introduction of Automatic Program Repair in Bloomberg" by Kirbas et al., shares experiences in integrating automatic repair into industrial practice. It represents an academic–industrial collaboration, where researchers from four universities have worked with Bloomberg engineers to allow for integrating high-quality automatic fixing of the Bloomberg code base. The article also discusses real-life lessons in bridging the academia–industry divide via shared practices derived from research—in this case, research on automated program repair.

known, there may be various fix locations in the program that can be changed to make the manifestation of the error disappear. Virtually all program-repair techniques thus begin by narrowing down the program source to a typically imperfect set of such likely locations.

Beyond that, program-repair techniques vary widely in how they modify the program source code to address the defect in question. Some approaches target particular bug classes and thus make use of a small set of candidate templates or transformations that may address bugs in that class (such as early work addressing buffer overflows). This approach is broadly generalized by work on search-based program repair (see, e.g., Le Goues et al.'s article in 2012[7]), which views automated patch construction explicitly as a search-space representation and navigation problem. Repairing a program thus amounts to navigating a search space of templated edits via a biased heuristic or random search. Such an explicit search approach must be guided by an objective function, often constructed in whole or part by the existing tests: a repair is found when a patch causes all tests to pass, including those that initially failed (indicating the bug).

A parallel line of work has focused on using semantic analysis to construct patches, by turning over the problem of bug repair to constraint solving and program synthesis, as described in the work of Nguyen et al.[8] This type of work infers a repair constraint that a patched program should satisfy to meet the given correctness specification, such as passing a given set of tests. Solving the constraint, typically via program synthesis, constructs a piece of replacement code that satisfies that specification. The replacement code can then be used as a patch. Constraint solving can encode other questions in the patch-construction process as well, like "What is the 'smallest' change in the program that will allow it to pass the given tests?" or even make suggestions on what the patch code might look like based on the code that is being repaired.

Alongside both of these families of techniques, it is important to note the growing, and perhaps unsurprising, role of machine learning techniques in automated repair. To date, learning-based techniques have been demonstrated to be usefully complementary to the other approaches. For example, models can effectively prioritize candidate patches that are generated by an enumerative search method.[9] However, this area of inquiry for program repair is rapidly growing, with researchers in both machine learning and software engineering/programming languages bringing their expertise to bear on the problem.

Regardless of how the patches are constructed, all automatic program-repair techniques are subject to a concern about "overfitting patches,"[10] or patches that address only the symptom represented by a failing test, rather than the true underlying cause of the error. Put plainly, just because a previously failing test now passes does not always mean the bug in question is truly fixed—and telling the difference, automatically, is an unsolved problem. Even human programmers sometimes commit incomplete or inaccurate bug fixes, by accident! Automatically fixed programs can be correct with respect to an incomplete program specification, such as a test set, but still be considered incorrect globally or as judged by a human maintainer. Much of the research in automated repair thus studies this aspect, with an attempt to generate correct patches despite operating over extremely partial specifications of program behavior (e.g., see the work of Shariffdeen et al.[11]). In the meantime, however, some practical deployment scenarios benefit from existing workflows in which patches or pull requests are reviewed by a developer before deployment, regardless of the source of the patch. Moving forward, we posit that patch quality will be a key issue in program repair, with manual or automated techniques used to enhance patch quality.

## Program Repair: A Snapshot

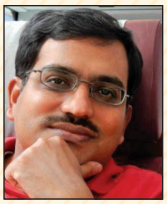
The successes of automated program repair, as the field stands today, have been significant. Successful techniques vary in terms of whether they address particular defect types or whether they aim to be more general to a wider variety of program properties that can be captured in a failing test. There has been tremendous progress in terms of enhancing generality of the techniques and scalability with respect to programs and search spaces. Modern research techniques of all stripes have reported successful results on programs of hundreds of thousands to millions of lines of code. Scalability to large search spaces (beyond simply to large programs) is important to allow the repair of complex, multipart bugs or programs that are significantly incorrect. Increasingly, such techniques begin to penetrate engineering practice.[12]

These results demonstrate that there exist pockets of opportunity where current approaches work well. However, significant room for improvement still remains, beyond the question of measuring and assuring patch quality. One challenge

## ABOUT THE AUTHORS

**CLAIRE LE GOUES** is an associate professor of computer science at Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, USA, where she is primarily affiliated with the Institute for Software Research. Her research lies at the intersection of software engineering and programming languages and focuses on the construction of high-quality software systems in the face of continuous software evolution, with a particular interest in automatic defect repair. Further information about her can be found at https://www.cs.cmu.edu/~clegoues for more information. Contact her at clegoues@cs.cmu.edu.

**MICHAEL PRADEL** is a full professor at the University of Stuttgart, Stuttgart, 70569, Germany. His research interests span software engineering, programming languages, security, and machine learning, with a focus on tools and techniques for building reliable, efficient, and secure software. Further information about him can be found at http://software-lab.org. Contact him at michael@binaervarianz.de.



**ABHIK ROYCHOUDHURY** is Provost's Chair Professor of Computer Science at the National University of Singapore, Singapore, 117417, Singapore, and the director of the National Satellite of Excellence in Trustworthy Software Systems, Singapore. His research interests are in program analysis, software security and trustworthy systems. Further information about him can be found at https://www.comp.nus.edu.sg/~abhik. Contact him at abhik@comp.nus.edu.sg.

**SATISH CHANDRA** is a software engineer at Facebook, Menlo Park, California, 94025, USA. His research has spanned many areas of programming languages and software engineering, including program analysis, type systems, software synthesis, bug finding and repair, software testing and test automation, and most recently, applications of machine learning to developer tools. Chandra earned his Ph.D. in computer science at the University of Wisconsin-Madison. He is an ACM Distinguished Scientist. Further information about him can be found at https://sites.google.com/site/schandraacmorg. Contact him at schandra@acm.org.

lies in expressive power or the variety of bug types that general techniques can handle off the shelf. In particular, most techniques struggle to construct or reason about changes that require multiple or significant changes to the program source. Although techniques can handle large programs, effective fault localization—the task of identifying the source locations to try to change—continues to challenge scalability for certain kinds of bugs and programs. Similarly, there exist continued challenges in effective engineering to apply these techniques to large systems since most repair approaches require expensive and complex recompile-test-and-check loops.

The three articles chosen in this special issue capture a snapshot of the current state of the field of program repair and point out possible future directions. See "New Repair Architectures, Techniques, and Practices" for summaries of the advances that it states as well as the challenges that remain. This will allow the reader to gain a critical appraisal of the repair technologies, as they stand today.

### Acknowledgment

This work was partially supported by the European Research Council under grant 851895 and by the German Research Foundation within the ConcSys and Perf4JS projects.

### References

1. R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Reading, MA: Addison-Wesley, 2003.
2. C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. doi: 10.1145/3318162.
3. A. Marginean et al., "SapFix: Automated end-to-end repair at scale," in *Proc. Int. Conf. Softw. Eng., Softw. Eng. Pract. Track (ICSE-SEIP'19)*, 2019, pp. 269–278.
4. J. H. Perkins et al., "Automatically patching errors in deployed

software," in *Proc. 22nd ACM Symp. Oper. Syst. Principles (SOSP'09)*, 2009, pp. 87–102. doi: 10.1145/1629575.1629585.
5. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf. (ATC'12)*, 2012, pp. 309–318.
6. B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. Int. Conf. Softw. Eng. (ICSE'13)*, 2013, pp. 672–681. doi: 10.1109/ICSE.2013.6606613.
7. 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, 2012. doi: 10.1109/TSE.2011.104.
8. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. Int. Conf. Softw. Eng. (ICSE'13)*, 2013, pp. 772–781. doi: 10.1109/ICSE.2013.6606623.
9. F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Principles Programming Languages (POPL'16)*, 2016, pp. 292–312. doi: 10.1145/2837614.2837617.
10. 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 Analysis (ISSTA'15)*, 2015, pp. 24–36. doi: 10.1145/2771783.2771791.
11. R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proc. 42nd ACM SIGPLAN Symp. Programming Language Des. Implementation (PLDI'21)*, 2021, to be published.
12. J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," in *Proc. ACM Programming Languages (OOPSLA'19)*, 2019, pp. 1–27. doi: 10.1145/3360585.



*Digital Object Identifier 10.1109/MS.2021.3082586*