Automated Bug Fixing

An Interview with Westley Weimer, Department of Computer Science, University of Virginia

and

Martin Monperrus, University of Lille and INRIA, Lille, France

by Walter Tichy

Editor's Introduction

On September 9, 1945, U.S. Navy officer Grace Hopper discovered a moth between the relays of the Harvard Mark II computer she was working on. She documented the incidence in her diary as "first actual case of bug being found." The term "bug" became quite popular. While hardware bugs are relatively rare, buggy software seems to be all around us. Hardly a week passes that we're not asked to update our PCs or telephones, because a feature doesn't work, data is lost or corrupted, a security hole has been discovered, or an applications crashes. While new and improved features may be welcome, most updates deal with bugs. So serious is the problem that large organizations such as Microsoft, PayPal, AT&T, Facebook, Google, and Mozilla offer bounties ranging from $1 to $100,000 for newly discovered bugs and their fixes.

Peter Newman, in his famous column "Risks to the Public" in ACM SIGSOFT Software Engineering Notes, has been documenting computer-related risks for decades. It would take a page just to list the categories; they include aviation accidents, space flight disasters, insurance fraud, telephone outages, public transport accidents, defective medical devices, security vulnerabilities, and stock market outages. Much could be learned from analyzing these bugs, but certain problems reappear again and again. To help find bugs, software engineers have developed inspections and testing methods. Test harnesses automatically subject software to thousands of tests. Research has uncovered correlations between bug-prone software modules and code complexity, test coverage, code churn, even the structure of the organization that produced the modules. These correlations can be used to identify bug-prone software and test it extra thoroughly. Yet we seem unable to get rid of bugs.

Enter a radically new idea: How about getting the computer to fix the bugs, automatically?

First the positive side: Fixing bugs manually is expensive, time-consuming, and unpleasant. Automatically repairing them might save us from the many causes of bugs, including misunderstandings, lack of time, carelessness, or plain old laziness. Automation might also produce bug fixes more quickly, especially if maintainers are faced with an overwhelming number of bug reports and must triage them. Even if only a portion of the bugs could be fixed automatically, doing so would be beneficial, because it would lead to better software more quickly, at lower cost. User frustration and losses due to bugs would be reduced.

The negative side is this is a bold idea, perhaps too bold. It goes against the famous halting problem, which asks whether an arbitrary program will stop or run forever. Alan Turing proved in 1936 that no general algorithm exists that can solve this problem for any program-input pair. So an automatic bug fixer can't even determine whether a fix will halt. Further, Rice's theorem states for any non-trivial property there is no effective way to determine whether a program computes a partial function that possesses that property. For instance, it is not possible to determine for any program, whether it recognizes a particular language. This means we can't write a general tool that determines whether a fix works. These are sobering, fundamental limitations for any automatic bug-fixing program. Somewhat less critical, but still serious, is the following: Bugs manifest themselves when execution reaches certain defective locations in a program. To test for the bug, execution must be directed to those locations. This means a number of variables in if-statements and loops on the path to the desired location must be set just right to reach that spot. This is an instance of the satisfiability problem, which is NP-complete. NP-completeness implies the work required to find the desired settings of variables grows exponentially with the number of such variables. Taking these theoretic limitations into account, most computer scientists would conclude automatically fixing bugs is a hopeless cause.

Yet in the past 10 years, a number of young scientists have taken on exactly that cause. It turns out that if one lowers expectations, a lot can be done. First, automated program repair does not promise to fix just any bug. Only small individual locations are fixable; bugs that require alterations of multiple locations are too hard, at least for now. Second, the technique requires a supply of test cases. Some of these test cases tickle the bug and fail. The other test cases do not fail; they represent desirable behavior that must be preserved. The automatic bug fixer must mutate the program in such a way that the failing test cases stop failing, while the non-failing test cases continue to pass. Note the specifications for the software are not provided; test cases are a substitute, which, as we well know, can show the presence but not the absence of bugs.

The following interview with two experts on automated program repair will discuss the approximations currently in use and how far they can take us.

Walter Tichy
Associate Editor

Automated Bug Fixing

An Interview with Westley Weimer, Department of Computer Science, University of Virginia

and

Martin Monperrus, University of Lille and INRIA, Lille, France

by Walter Tichy

Ubiquity: You have developed GenProg, an algorithm that is supposed to fix program bugs automatically. How can this work? Aren't you up against several undecidable problems at once?

Westley Weimer: GenProg explores the space of possible patches until one is found that repairs the defect while retaining required functionality. At each step of the way we make approximations to make this otherwise-impossible problem feasible: we only consider small patches, we only patch parts of the program likely to contain the fault, we assume the program contains the seeds of its own repair (we never invent new code), and we use test cases as a proxy for correctness.

By analogy, GenProg is quite a bit like an introductory CS student facing a bug. First you might put in "print" statements to narrow down where the crash is. Then you randomly delete and copy nearby lines until the result passes all of the tests. At its heart, GenProg has automated the failings of a new CS student.

Ubiquity: This never worked for me! I quickly found out that reasoning was more effective than thoughtless trial and error. Why would trial and error work here?

WW: Reasoning *is* often more efficient, just as symbolically analyzing a function may be more efficient than using simulated annealing to optimize it. But GenProg, just like simulated annealing or Newton's method, can work quite well given enough resources—and machine resources are less expensive than expert human time. GenProg uses the program's test cases like simulated annealing uses its objective function: to guide and constrain the search, keeping around those changes that pass more tests. In addition, analyzing the program's behavior on its

test cases tells us where to make modifications: we prefer changes to statements implicated on failing runs but not on passing runs. We use those same test cases to validate a candidate patch, passing along to developers only those patches that cause the program to pass every available test case. So GenProg is not purely blind: it targets the search to implicated program areas and it prefers changes that pass more of the test cases.

Ubiquity: How complicated are the defects and fixes that GenProg handles? Do they involve several program locations, are they one-liners, or simple fixes to expressions?

WW: One advantage of GenProg is that it can be applied to many types of defects and does not require the defect to be known in advance. The majority of fixes produced by GenProg are "one-liners"—as are the majority of fixes produced by humans. In experiments, GenProg fixes about one-third to one-half of real-world defects in off-the-shelf legacy programs. It is much more successful when the fault can be localized to a single file. GenProg can change individual expressions (and we can repair assembly programs and binary executables by changing individual instructions) but it typically operates at the level of statements. For source code, statements provide a good balance: they are expressive enough to construct many fixes, but they are small enough to keep the search space limited.

Ubiquity: How does GenProg generate fixes?

WW: The two key choices are "what statement should we edit?" and "what should we change that statement to?" The first question is known as *fault localization* and has been well studied in software engineering. We call the second question *fix localization*, and it relates to the "shape of the fix" (e.g., "bring in a null check" or "insert a function call" or "insert an assignment statement"; Martin Monperrus is an expert in this area). Various versions of GenProg have used genetic algorithms, brute force approaches, and even adaptive searches to prioritize the exploration of the fault space and fix space, with the goal of enumerating a real repair early.

Ubiquity: What are some of the other approaches for generating fixes?

WW: There are now more than 20 research groups around the world working on program repair, so I won't be able to mention every one. Some popular approaches include PAR, which uses insights from historical human-written patches to shape its fix space; ClearView, which constructs run-time patches for security-critical systems programs and has survived Red Team competitions; and AutoFix, which uses contracts and pre- and post-conditions to find provably-correct repairs. In addition, there are a number of approaches that target particular classes of defects, such as AFix, which repairs atomicity violations in concurrent programs.

Ubiquity: Please summarize how fault localization works.

WW: Fault localization, such as the influential *Tarantula* technique of Mary Jean Harrold and her collaborators, usually produces a list of program elements (such as statements) ranked by their likely implication in a fault. Fault localization can answer the question "Given that I am failing test case X but passing test case Y, where is the bug? What part of the source code should I change if I want to pass test X?" A simple approach to fault localization is "print-statement debugging": after instrumenting the program to print out each statement visited, the statements visited on failing test X but not on passing test Y are likely to relate to the bug. Importantly, most bugs can be localized to a small fraction of the program's source code – a few hundred implicated lines even in a million-line program, for example. Automated program repair techniques then use the ranked list of implicated statements to decide where to make edits next.

Ubiquity: How can you be sure that GenProg actually fixed a bug?

WW: How can you be sure that humans actually fixed a bug? For example, a recent study of twelve years of patches by Yin *et al.* found that 15–24 percent of human-written fixes for post-release bugs in commercial operating systems were incorrect and impacted end users. We like to imagine that human developers are some sort of gold standard, but code churn and reverted patches are significant issues, and humans themselves are not always correct.

All GenProg patches are correct by definition in the sense that they pass all available test cases. Others and we have also evaluated GenProg's patches using held-out workloads, held-out additional exploits, and human studies. GenProg patches are not as maintainable or acceptable to humans as are human-written patches, but GenProg patches augmented with synthesized documentation can be.

Formally, fixing a bug might involve bringing an implementation more in line with its specification, and the specification may not be known or may change over time in the face of changing requirements. In my opinion, providing evidence that increases developer confidence in machine-generated patches is one of the next big research questions in this area.

Ubiquity: Since GenProg searches for possible patches without human guidance, surely some of the candidate patches must have been comically bad. Can you share a few of the early failures of automated program fixing?

WW: We sometimes use the analogy that GenProg is like a young child: "you only said I had to get in the bathtub, you didn't say I had to take a bath." Like any search optimization problem, GenProg maximizes the notion of correctness you give it. One of our earliest examples was a webserver with a remote exploit related to POST functionality. We did not include any test

cases for POST functionality, so GenProg quickly found a simple repair that disabled it. On the more extreme end, when we fail to sandbox correctly, GenProg has produced candidate "repairs" that find and overwrite the expected test case answers with blanks and then exit immediately, appearing to pass all tests. Our most noteworthy failed "repair" revived unused code to email the original developers—and since we were running thousands of variants using cloud computing, that was enough email for Amazon to flag us as a spammer.

Ubiquity: One of your publications claims that it can fix defects for eight dollars apiece. How did you come up with that?

WW: We are firm believers in reproducible research. With modern, publicly available cloud computing, it is possible to pay services such as Amazon's EC2 or Microsoft's Windows Azure for CPU-hours. We wanted to make a business case for GenProg: If it costs us $50 to fix a bug and it costs humans $20 to fix a bug, there is not much of a reason to use GenProg. But if it costs GenProg $8 to fix a bug, that might be worthwhile, even if the resulting patch is of slightly lower quality. To arrive at the $8 number we ran all of our experiments on publicly available cloud computing and recorded the total compute cost (including the cost of failed repairs) and divided by the number of successful repairs. Any researcher or company can reproduce our results for the same amount (actually, less— cloud computing prices continue to plummet), and all of our benchmarks and algorithms are available via virtual machines at http://genprog.cs.virginia.edu.

Ubiquity: We'll switch to our second interviewee, Martin Monperrus. Some of the automatic patches are strange and appear complicated—not what a human would right. If these patches are difficult to understand, isn't that a problem for future maintenance?

Martin Monperrus: Yes and no. Yes, if the patch is meant to be a permanent solution. One does not always have the resources to fix bugs as fast as they are discovered. In those cases, an automatically synthesized patch is a temporary solution, for instance to avoid crashes. This patch, waiting for human validation or improvement, is indeed valuable regardless of its appearance.

The scope of automatic repair is not only to compete with human-written patches. Automatic repair can also provide patches in situations when there really is no time for humans to handle the defect.

Ubiquity: GenProg uses code snippets from the faulty program and mutates them rather than "invent" its own. Is that a smart idea or a limitation?

MM: This "closed world hypothesis" (no invention of new code) is very smart. According to our experiments, one out of ten commits by humans does not add any new code; it merely rearranges existing code. Exploiting the existing software is key for simplifying the synthesis of patches and for reducing the time to find a fix.

That said, there are machine-generated, one-line patches that consist of new code. For instance, if the bug is in an arithmetic expression, there are techniques to find a patch without assuming a closed world.

Ubiquity: Heuristic search is a technique for situations where you don't know any better. Are there alternatives to searching for patches?

MM: In repair, there are many interwoven kinds of search. Wes already mentioned the difference between searching for the statement to edit and searching for changes of that very statement. There are alternative techniques for both. For instance, in certain cases, the search for the statement to edit can be done using symbolic execution.

To me, there is no competition between the alternatives. I rather view them as complementary:

Depending on the kind bug, the domain, or the size of the system to be repaired (and many other factors), any alternative may turn to be more or less appropriate. The research community is working on characterizing those differences.

Ubiquity: Are the defects that can be repaired mostly blunders, i.e. stupid or careless mistakes?

MM: Whether a bug comes from a careless programmer or an ambiguous specification does not matter: if it shows up in production, it must be fixed. Automatic repair is agnostic about the cause of the bug.

Regarding the complexity of patches, simple patches sometimes take a very long time to be found by expert developers. Indeed, there is no obvious and direct relation between the complexity of a patch and its difficulty. This is the same for automatic repair: there exist small fixes that are impossible to find automatically and large ones that are rather simple to discover.

Ubiquity: Are there any defect classes and their fixes, such as "off by one" or "forgotten null check"?

MM: I have a pretty wide definition of "defect class": It is a family of bugs that have something in common: They can share the same root cause (e.g., a particular kind of programming mistake, such as an off-by-one defect, or a misunderstanding of a specification), the same symptom (e.g., an exception) or the same kind of fix (e.g., changing the condition in an if-statement). Of course there is a relation between the root cause and the kind of fix. For instance, a null pointer exception can be fixed by adding a null check.

However, the relation between the symptom and the fix can be a complex one. For instance, the off-by-one errors can be fixed at a memory allocation point, in a loop condition, or even by silencing the error at runtime. On the one hand, many different bugs can be fixed in the same manner; on the other hand, it happens that the same bug can be fixed in diverse ways.

Ubiquity: Test cases are critical. Should these test cases be white box or black box, i.e. should they be written with or without knowledge of the test subject's implementation?

MM: I completely agree when Wes says that repairing a bug means, "bringing an implementation more in line with its specification". What is really critical for repair is a piece of specification, even partial and incomplete. Test cases are one such specification; other practical ones include runtime assertions and contracts.

From the viewpoint of repair, whether the test is white box or black box does not really matter. What matters is that the test case specifies the important points of the input domain. However, the question touches the interesting link between bugs and program boundaries. Let's call the latter "APIs" (application programming interfaces). From the client's viewpoint, a failing test case is a black-box test. However, additional finer-grain white-box test cases can be of great help for driving the diagnosis and repair process.

Ubiquity: I applaud the attempts to make results reproducible by using benchmarks. Benchmarks accelerate progress. However, the selection of benchmarks introduces a potential bias. How should repair techniques be evaluated?

MM: Repair techniques should be evaluated with carefully crafted benchmarks containing meaningful bug instances. There must be a correspondence between the defects of the benchmark and the defects targeted by a given repair technique. For instance, applying a repair technique for null pointers to an off-by-one bug benchmark is meaningless. Similarly, mixing completely different bugs in the same benchmark makes little sense.

To avoid bias, the best method is to carefully characterize the problem. Benchmark providers must clearly describe the targeted defect classes, selection criteria, and frequency of each defect class. Inventors of repair techniques must make sure they evaluate their technique with the right benchmark.

Ubiquity: Is there any hope that repair robots can guarantee the correctness and completeness of their patches?

MM: The correctness and completeness of automatic patches directly depends on the correctness and completeness of the specification. For programs where there exists a correct and complete formal specification, my answer to your question is "yes". For the remaining 99.99 percent of programs, I don't think so.

However, there is hope that repair robots will accumulate such an impressive track record of successful and long-lasting patches that we trust them. We already place confidence in critical systems that come with no guarantee.

Ubiquity: How far is this technology from practical application? What has to happen in research to make it practical?

MM: It depends on the defect classes. Some repair techniques are almost ready for industrial use; others need more time to mature. For instance, automatic repair of off-by-one errors is not far from practical application.

More generally, for this research to have practical impact, I would say it is like most software engineering research results: we need entrepreneurs to transform a prototype into a product and create a new market.

WW: Farther away than I might like. Despite isolated incidents (e.g., Dr. Sigrid Eldh at Ericsson has evaluated GenProg on commercial code and defects) our work remains primarily a research project.

I see three primary barriers to practicality: GenProg has to be able to read in your code, GenProg has to be able to compile your code, and GenProg has to be able to run your tests (or check your invariants, etc.). In practice, real-world code is often written in multiple languages, uses bootstrapping or third-party libraries that complicate analysis, only compiles or runs on special machines, and may not have a convenient testing interface that can handle ill-behaved programs (like the ones mentioned above that overwrite the expected answers). It is often easier for GenProg to find a way around the intent of a weak test case than to make a repair that truly pleases humans, but few have the resources to write even more test cases.

Automated program repair holds out the promise of being a rising tide that lifts all boats. Software maintenance is a dominant cost, and programs ships with known and unknown bugs. If we could handle even the "easy half" of defects automatically we would free up developer

effort to focus on tougher bugs and new feature additions. Significant research challenges going forward include fixing more bugs, fixing more types of bugs, producing high quality fixes, and giving humans confidence in those fixes. It's an exciting time for this research topic!

Closing Remarks

Automated program repair has amazing potential: It could reduce time and cost of fixing bugs, lower user frustration with software, perhaps even avoid accidents. Compare this to recalls for cars: It takes months and months of time and can cost car manufacturers billions. If software repair is cheap and fast, then there will be less temptation to put it off (perhaps preventing disasters like those caused by GM's faulty ignition locks).

At this point in time, automated bug fixing is still firmly in the research area. Single-line fixes will not be enough. An analysis of three large open source systems (Firefox, Mylyn, and Evolution)[1] found the median size of a change ranges from 24 to 62 lines, with the mean being in the hundreds. Thus, automatic repair will have to do handle multi-line fixes to be useful. We will see whether researchers can scale automated repair to five or ten lines or beyond, at which point software maintenance will change fundamentally.

The deeper philosophical issue is perfection. We live in an imperfect world, have imperfect knowledge and an imperfect brain, so our creations will always be imperfect. Near perfection can be achieved, but only in rare moments, such as in a work of art, a mathematical theory, or an elegant algorithm. Automated software repair will certainly not lead to perfect software. The bugs have to be found before they can be fixed, so at least some users will be exposed to them. Once found, only a portion can be fixed automatically. Hypothetically, even if we could detect and fix all bugs before releasing software to users, we cannot anticipate all the circumstances in which the software will be used. For example, the first texting program I used was "talk". It ran on ARPANET in the seventies. At the time, I did not occur to me that one day people would text while driving and cause accidents. I would not have thought of putting a texting-while-driving-preventer into the software. Similarly, the inventors of the Internet did not anticipate attacks by malicious users; too precious were the new capabilities. As a consequence, Internet protocols are vulnerable to fraud and espionage to this day. These examples illustrate the unpredictable paths of inventions may lead to new types of flaws, even if the inventions worked perfectly on day one.

Will automated software repair cause new types of risks because of an over-reliance on automatic fixes? Will machine-generated corrections lead to unintelligible and unmaintainable code and therefore accelerate the decay of software? As always, we will address these issues once they manifest themselves.

[1] Thomas and Murphy, How effective is modularization?, In Oram and Wilson (eds.), *Making Software*. O'Reilly, 2010.

Reading List

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for $8 Each. International Conference on Software Engineering (ICSE) 2012, 3-13.

Claire Le Goues, Stephanie Forrest, Westley Weimer. Current Challenges in Automatic Software Repair. *Software Quality Journal* 21, 3 (2013), 421-443.

Westley Weimer, ThanVu Nguyen, Claire Le Goues, Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. International Conference on Software Engineering 2009, 364-374

Martin 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. International Conference on Software Engineering, 2014, 234-242.

Matias Martinez, Westley Weimer, Martin Monperrus. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. International Conference on Software Engineering, 2014, 492-495.

About the Author

Walter Tichy has been professor of Computer Science at Karlsruhe Institute of Technology (formerly University Karlsruhe), Germany, since 1986. His major interests are software engineering and parallel computing. You can read more about him at www.ipd.uka.de/Tichy.

DOI: 10.1145/2746519

