The Evolution of Automated Software Repair

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest *Fellow, IEEE,* Westley Weimer

***Abstract*—GenProg implemented a novel method for automatically evolving patches to repair test suite failures in legacy C programs. It combined insights from genetic programming and software engineering. Many of the original design decisions in GenProg were ultimately less important than its impact as an existence proof. In particular, it demonstrated that useful patches for non-trivial bugs and programs could be generated automatically. Since the original publication, research in automated program repair has expanded to consider and evaluate many new methods, contexts and defects. As code synthesis and debugging techniques based on machine learning have become popular, it is informative to consider how views on perennial issues in program repair have changed, or remained static, over time. This retrospective discusses the issues of repair quality (including the role of tests), use cases for automated repairs (including the role of humans), and why these approaches work at all.**

***Index Terms*—Automatic programming, corrections, testing and debugging, evolutionary computation**

I. Introduction

The 2012 article "GenProg: A Generic Method for Automatic Software Repair" described GenProg, a technique that used genetic programming (GP) to automatically generate patches for bugs in programs, identified by test cases. GP is a stochastic search method inspired by biological evolution that discovers computer programs tailored to a particular task [1], [2]. It relies on computational analogs of biological mutation and crossover to generate new program variations; in GenProg, we referred to these variations as *variants.* A user-defined fitness function evaluates each variant. GenProg uses the provided test cases to evaluate variant fitness. Individuals with high fitness are selected for continued evolution in an iterative search process. The process is considered successful when it produces a high-fitness variant according to the user-defined function. For GenProg, this means a program that passes all tests encoding required behavior, including those that expose the bug.

That TSE'12 article was an invited extension of a 2009 Distinguished Paper at the International Conference on Software Engineering (ICSE), "Automatically Finding Patches Using Genetic Programming" [3]. The TSE article unified and expanded on previous publications [4], [5], especially in terms of the evaluation, including:

- New bugs and programs. The article reported on bugs in five additional programs compared to the previous evaluations, doubling the total amount of repaired code evaluated to over 100k LOC. These new programs included four new types of errors.
- Closed-loop repair. The TSE paper described, and provided a proof-of-concept evaluation of, a closed-loop repair system integrating GenProg with anomaly intrusion detection.
- Repair quality. The evaluation significantly expanded the original publication's treatment of repair quality. This included both a manual/qualitative assessment and a quantitative study using indicative workloads, fuzz testing, and variant bug-inducing input.

Manuscript received January 2025.

GenProg was impactful because it demonstrated the feasibility of automating program transformation for bug repair on large (for the time) corpora of real-world open-source software used in production. Automating this type of programming had been proposed, particularly in the evolutionary computation research area, but never explored seriously for production code. Over the subsequent decade or so, Automatic Program Repair (APR) has developed into a significant research subfield at the intersection of program analysis and software engineering. This work includes an adaptive radiation of methods for finding bugs and generating repairs, some of which have been integrated into industrial deployments [6], [7]. The recent development of transformer-based language models and generative AI, as applied to code, has accelerated interest in this type of work.

In our view, many of the details of the GenProg algorithm, as described in the article, are less important to its impact than its status as an existence proof. Many design decisions were under active investigation at the time, e.g., in the same year that the TSE article was published, we also published a more efficient representation, which evolved edits to programs, or patches [8], rather than entire ASTs. This has turned out to be a more enduring design choice.

However, at least two design choices *were* significant. First, GenProg targeted a widely-used, general-purpose programming language (C). This, coupled with the fact that its program analysis strategy was relatively lightweight, allowed GenProg to scale to nontrivial modules and programs. This emphasis on real open-source code endures in the APR research literature, which continues to emphasize empirical studies, evaluations on corpora of bugs in open repositories, and comparisons among methods.

Second, GenProg relied exclusively on test cases to enforce correctness. Testing was (and remains) the most common Quality Assurance technique in development practice. Relying on tests rather than any kind of formal correctness specifications allowed GenProg to apply to a wide variety of programs, bugs, and errors. Using tests as proxies for correctness specifications diverged strongly from common research wisdom and practice at the time. It was a controversial design decision, and led to significant ongoing debate and study. However, the use of tests to inform and guide APR remains a de facto standard in the program repair and transformation literature. This includes techniques that, like GenProg, use test cases to guide repair directly and more recent approaches from the ML/AI community which use tests to evaluate repairs in response to issue or bug reports (e.g., [9]).

II. Evolution of Repair Quality

A major focus of the experiments unique to the TSE'12 article was *repair quality*. Although using tests to guide repair allows techniques like GenProg to apply broadly, those tests are also, by their nature, partial. It is always possible to construct a program that satisfies a partial specification but that does not generalize to the (unwritten) full specification.

Repair quality, often referred to as overfitting [10], has been a major focus of subsequent work in program repair, both on its own and as a component of evaluations of new techniques. Generated tests, separate held-out tests, and fuzzing campaigns have been used to evaluate repair quality, usually as an adjunct to manual comparison to the developer-written "gold standard" fix for a historical bug. This standard is more restrictive than the one we applied in this article, when such experimental norms had yet to be established, and as we argue below is likely overly restrictive. Modern evaluations, therefore, define "correct" differently from the TSE paper, even though both types of assessment are effectively manual. Both approaches, however, acknowledge the reality that tests provide only partial arguments for quality. This is important to bear in mind when evaluating and making claims about any new technique for program transformation—whether based on more formal reasoning, or on ML-based approaches.

That said, in the qualitative analysis of produced patches, we carefully selected bugs that we considered illustrative; the natural (or developer-provided, if available) fix; and the fix produced by GenProg, if it differed. This is more subjective, and less principled, than a strict comparison to a developer-provided gold-standard patch, and it is impractical at the scale of modern evaluations. However, there are benefits to such narrative assessment of produced patches, especially for those that do not perfectly reproduce the text of a historical fix. For example, a GenProg repair to `openldap` fixes a denial-of-service attack by removing a buggy loop that sanity-checks very large request tags. This repair has the side effect of limiting the range of request tags to 127, which is different from the gold-standard repair (which removes extraneous assertion failures in that same loop). However, since `openldap` only has 30 defined tags, the patches produce identical behavior in practice. Such narratives add color as to what the technique can and did do, and it acknowledges the fact that there are an infinite number of ways to fix any particular defect. This theoretical diversity also manifests in practice, including when multiple experienced developers repair the same defect [11].

These results, and the ongoing challenges faced by researchers in evaluating program transformation techniques, highlight a need for more nuanced studies of repair quality, and standards for how to evaluate it experimentally. One recurring observation was that the evaluation standards for generated repairs often varied with the use case: a patch that would be deployed automatically might be considered differently from one that would serve as informal guidance to a human.

III. Evolution of Use Cases

Over the years, APR efforts have considered multiple use cases for generated patches. One primary dimension of variation is the degree of human involvement. We consider a highly-manual use case and a fully-automated one.

Perhaps surprisingly, our original model envisioned candidate patches being presented to developers as suggestions, perhaps via IDE integration. This was the use case considered in precursor work to GenProg [12]: Proposed patches would be shown to developers as parts of bug reports, and even incorrect patches might reduce development time by providing ideas. Although that use case was quite speculative in 2006–2009, the rise of cloud computing led to the widespread adoption of continuous integration continuous delivery (CI/CD) techniques. The ensuing tighter push, test, commit, review cycle more naturally placed humans in front of small, proposed changes. Human inspection as part of software evolution has a rich history, but CI/CD has increased acceptance of a faster feedback loop of small changes.

One aspect meriting particular attention is an apparent, very recent, LLM-driven shift in developer tolerance for tools that are explicitly not guaranteed to be correct. Informally, in prior decades it was widely accepted that industrial developers were intolerant of false positives [13], [14]. In our recent conversations with industrial developers, however, they appear to be more willing to receive and fix up slightly-incorrect suggestions from synthesis tools like ChatGPT. This remains an important use case and research topic, with a 2024 study finding that access to correct APR suggestions increases the odds of debugging success by a vastly larger margin as compared to having access only to tests. Access to overfitting suggestions decreases the odds of debugging success, but it hurts less than having good suggestions helps [15].

A second family of use cases emphasizes the automated deployment of patches without a human in the loop. The TSE'12 article considered a case study of a long-running webserver deployment, imagining a scenario in which inputs that triggered an anomaly detection system would pause the system, be passed to GenProg to produce a patch, and then that patch would be deployed automatically on future inputs.

In the scenario, the webserver was running a version of `php` vulnerable to a remote exploit against the `str_replace` function, and 130,000 historical HTTP requests and 12,000 historical PHP requests were used. We first assessed the time cost associated with pausing the webserver to conduct repairs, determining that 2% of requests would be delayed. We also assessed the requests lost to repair quality, as when a GenProg repair closes the vulnerability but disables single-character string replacements. In this setting, the fraction of PHP requests that differed by even one byte was not statistically different from zero. One lesson was that functionality-deleting repairs do not always have an impact on the user experience.

This fully automatic use case has been less emphasized in the research since the article was written. The latter case study especially highlights that the role of repair quality is subtle, and its importance can vary strongly by use case. Deleting functionality to automatically prevent the exploitation of key vulnerabilities may be acceptable in some cases, such as when a human operator is unavailable and the vulnerability particularly severe. Many bugs and vulnerabilities appear on less commonly-executed code paths; our case study suggests that removing such paths may sometimes enhance security, while not meaningfully impacting user experience. However, these considerations are necessarily contextual, and thus much more difficult to evaluate in the types of large-scale empirical settings that have become common in program repair literature since.

Use case and evaluation concerns are timeless. The overarching question of where and to what degree human developers should be 'in the loop' remains, and our answers to this question are changing quickly as ML-based methods mature.

IV. Evolution of Software

It is unsurprising that text prediction methods that work so well for generating natural language should also work well for generating program code, which has much smaller vocabulary and simpler grammatical structure [16]. More surprising is the fact that approaches based on random mutation, such as GenProg and its successors, can succeed at all.

Independent of progress developing APR tools that can repair more bugs more correctly than the original GenProg, a separate line of work has asked why GenProg and its successors succeed as often as they do. A key assumption of the GenProg architecture is that the ingredients of a repair are likely to already exist in the program [17], which enables mutation operators to succeed that simply reorder, delete or copy code. This assumption is known as the *plastic surgery hypothesis* [18] and is an important enabler of the success of GenProg and other mutation-based methods. In addition, many researchers have observed that most bugs are "small" [19], which helps explain why GenProg's repairs, which usually consist of one- or two-line edits, are often sufficient.

Other work considers the hypothesis that software has certain properties that make it inherently "evolvable," i.e., amenable to random mutation and selection. As just one example, several studies [20]–[22] have measured *mutational robustness*, i.e., how likely it is that a random mutation will change the observed behavior of the program. These studies have shown that, in programs written in different languages and at different abstraction levels, software is highly robust to mutation (e.g., $\approx 30\%$ of mutations are non-harmful for source code), which is comparable to observations made in some biological systems [23]. In biology, mutational robustness is believed to be a key enabler of evolution, because it allows the search to explore the adaptive landscape more widely without fatal consequences, and thus improves the chances of discovering important innovations [24]. Why this robustness exists in software, and what its effects are, are interesting and unanswered questions.

In SE terms, mutational robustness can be thought of as "neutral mutations considered helpful," in contrast with the subfield of mutation analysis, where neutral mutations complicate the interpretation of mutation adequacy scores and are usually considered negatively, as evidence of either a missing test or semantic equivalence [25]. In many cases, however, mutations can produce functionally-correct code improvements, even if the result is not strictly equivalent semantically. For example, a mutation that changes run-time behavior but still produces correct results might be considered semantically distinct but functionally correct. This principle was leveraged in follow-on work to GenProg that optimized quality properties, such as energy efficiency or run-time, by discovering variants that behave differently but acceptably [26], [27]. Beyond mutational robustness, other aspects of software have been shown to resemble natural biology as well, including epistasis (interaction among genes), neutral landscapes, and bimodal fitness distributions [27], [28].

More macroscopically, modern software development practices incorporate the three key mechanisms of Darwinian evolution: variation, selection, and inheritance. Variations, random or otherwise, are introduced whenever a developer or tool changes a line of code. They are also introduced through an analog of the biological process of crossover, whenever two libraries, code snippets, or modules are combined from different sources. Selection and inheritance occur each time a piece of software is copied — successful software is copied frequently and becomes prevalent in the software ecosystem, and unsuccessful software fades away. Modern tools and development practices, including everything from Stack Overflow to GitHub to A/B testing of interfaces to continuous integration all reinforce and accelerate this dynamic.

V. Conclusion and Looking to the Future

APR has evolved significantly, from the traditional GenProg that used testsuites, GP, and focused on C to techniques that use many methods (formal specifications, program synthesis, machine learning, etc.) and apply to many languages. APR has been transformed from academic research to industrial deployment (e.g., Facebook's SapFix [7] and Getafix), and was a precursor to AI-driven code suggestion tools (e.g., Codex, Copilot).

Looking back to GenProg's theoretical underpinnings, given the evidence that the mechanistic drivers of evolution are all present in modern software practices, important questions include: Can we measure these effects? How do evolutionary dynamics affect or constrain the overall trajectory of software development? What are best practices for leveraging these evolutionary dynamics in the context of human- or AI-driven software development?

While the potentials of APR are exciting in the modern area of AI, there are still many challenges that need to be addressed. For example, there is interest in using APR techniques to debug AI models, potentially addressing issues of AI trustworthiness and safety. While such applications (e.g., repairing deep neural networks) are exciting and have the potential for significant impact, a number of domain-specific challenges remain. These include scalability (AI models are large), repair quality (while software behavior is often captured by tests or specifications, AI is often defined by a training set that is not as easily captured by traditional methods), and use case (AI models are not easily interpretable, so how and how much the human should be in the loop is less certain).

Conversely, as AI continues to support code synthesis and debugging, careful attention may be merited for some use cases and defects. Some perceive that certain classes of defects, like zero-day vulnerabilities that often have few analogs in training sets, may be more difficult to solve with LLM-based approaches. In addition, the human-in-the-loop aspect of repair quality assessments and use cases is more critical with ChatGPT interfaces and IDE tools like Codex. We speculate that it will become increasingly important to train developers to assess candidate patches and synthesis tools.

Finally, just as extending APR to AI models stretches our notion of what a "program" is, years of work by many researchers have stretched the notion of what a "bug" is. In a recent article, we argued that the definition of a bug includes subjectivity and judgment, notions that also inform whether it should be fixed: "tests can fail for any number of reasons—flakiness, failed code style checks—that we do not ordinarily consider bugs. Further, there are almost always more bugs reported than can be reasonably handled given available resources, a fact baked into modern continuous-deployment pipelines and bug triage processes." [29] As APR is increasingly stretched from its original conception of passing test cases to more modern use cases of improving non-functional properties or supporting AI models, the question of *what we want software to be*—and how that differs from what it is now—remains central.

Acknowledgments

While many people were involved in the genesis of GenProg over many years of research and development, the authors are particularly indebted to Mark Harman, Martin Monperrus, and Andreas Zeller for substantial support around the timeframe of the 2012 TSE article. This work was supported over the years by several funding agencies, including: ONR MURI Program, DARPA, AFOSR, and NSF. We acknowledge the partial support of NSF (CCF2319131 CCF2211750), DARPA (N6600120C4020), ARPA-H (SP4701-23-C-0074), and the Santa Fe Institute.

References

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[2] S. Forrest, "Genetic algorithms: Principles of natural selection applied to computation," *Science*, vol. 261, pp. 872–878, Aug 1993.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 364–374.

[4] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM Research Highlight*, vol. 53, no. 5, pp. 109–116, 2010.

[5] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conference*, 2009, pp. 947–954.

[6] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, "On the introduction of automatic program repair in bloomberg," *IEEE Software*, vol. 38, no. 4, pp. 43–51, 2021.

[7] 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: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 269–278.

[8] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2012, pp. 3–13.

[9] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://openreview.net/forum?id=mXpq6ut8J3

[10] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Foundations of Software Engineering*. ACM, 2015, pp. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[11] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Foundations of Software Engineering*. ACM, 2017, pp. 117–128.

[12] W. Weimer, "Patches as better bug reports," in *Generative Programming and Component Engineering*, 2006, pp. 181–190.

[13] N. Ayewah and W. W. Pugh, "The google findbugs fixit," in *International Symposium on Software Testing and Analysis*, P. Tonella and A. Orso, Eds., 2010, pp. 241–252.

[14] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[15] H. Eladawy, C. Le Goues, and Y. Brun, "Automated program repair, what is it good for? not absolutely nothing!" in *International Conference on Software Engineering*. ACM, 2024, pp. 84:1–84:13.

[16] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *International Conference on Software Engineering*, 2012, p. 837–847.

[17] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th international conference on software engineering*, 2014, pp. 492–495.

[18] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Foundations of Software Engineering*, 2014, p. 306–317.

[19] C. K. Shriram, K. Muthukumaran, and N. L. Bhanu Murthy, "Empirical study on the distribution of bugs in software systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 01, pp. 97–122, 2018.

[20] E. Schulte, Z. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, pp. 1–32, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10710-013-9195-8

[21] B. Baudry, S. Allier, and M. Monperrus, "Tailored source code transformations to synthesize computationally diverse program variants," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 149–159.

[22] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Comput. Surv.*, vol. 48, no. 1, Sep. 2015.

[23] R. Sanjuán, "Mutational fitness effects in RNA and single-stranded DNA viruses: common patterns revealed by site-directed mutagenesis studies," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 365, no. 1548, pp. 1975–1982, 2010.

[24] M. Kimura, *The Neutral Theory of Molecular Evolution*. Cambridge: Cambridge University Press, 1983.

[25] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[26] J. Dorn, J. Lacomis, W. Weimer, and S. Forrest, "Automatically exploring tradeoffs between software output fidelity and energy costs," *IEEE Trans. on Software Engineering*, vol. 45, pp. 219–236, 2019, on-line version published Nov. 2017.

[27] J. Liou, M. Awan, P. Sulc, K. Leyba, S. Hofmeyr, C. Wu, and S. Forrest, "Evolving to find optimizations humans miss: Using evolutionary computation to improve GPU code for bioinformatics applications," *ACM Trans. Evol. Learn. Optim. (TELO)*, vol. 4, no. 4, 2024.

[28] J. Renzullo, W. Weimer, and S. Forrest, "Evolving software: Combining online learning with mutation-based stochastic search," *ACM Trans. Evol. Learn. Optim. (TELO)*, vol. 3, no. 13, pp. 1–32, Dec. 2023.

[29] D. G. Widder and C. Le Goues, "What is a 'bug'?" *Commun. ACM*, vol. 67, no. 11, pp. 32–34, 2024.