Test Cases Considered Harmful

Steven P. Reiss
spr@cs.brown.edu
Department of Computer Science, Brown University
Providence, RI, USA

Abstract

We argue that the use of test cases and extensive test suites to drive APR is harmful to the field and we push for something better. Such test suites are rare in practice. Moreover, using test cases for APR tends to skew how and where APR is used and bias experiments that attempt to validate its techniques. We suggest that other approaches to APR which do not assume such test suites are available are more appropriate. We also suggest developing new means for evaluating APR systems that do not depend solely on test-case-based examples. We challenge researchers to create such approaches and evaluations for a broad range of problems.

CCS Concepts

• **Software and its engineering → Software creation and management**; *Software development techniques.*

Keywords

Automatic Bug Repair

ACM Reference Format:

Steven P. Reiss. 2026. Test Cases Considered Harmful. In *Proceedings of Automated Program Repair (APR 2026)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Test cases have been widely used as the basis for Automated Program Repair (APR) [28] ever since GenProg [16]. They are still used today for fault localization, patch validation, and system evaluation. Using test cases has made it easy to experiment with APR through the use of spectra-based fault localization and by using passing test cases for patch validation. In addition, standard APR test suites based on fixing failing test cases provide a simple means of mechanically evaluating and comparing APR systems.

However, test cases do not serve APR well and have skewed their applicability, utility, and development. There are a number of reasons for this.

First, few systems have a test suite that can support APR. Using test cases for APR assumes the existence of a high-quality, comprehensive test suite. In practice, high-quality test suites are often unavailable. As suggested by [4, 14], developers often do not write a test suite containing a sufficient number of test cases or even write tests at all especially during initial development. We scanned all Java projects in the Merobase source repository [10] and found that of the 12,525 projects with 50 or more methods, about 58% of them have no tests at all. In fact, only about 13% projects had the density of tests per method as the minimum of the packages in Defects4J [12] used to evaluate APR techniques. This limits the utility of test-based APR.

Second, using test suites biases our evaluations. Many evaluations are based on Defects4J or its more recent extensions. Defects4J consists mainly of code for libraries. The code is not multi-threaded, rarely interactive, and not particularly complex. Real Java systems are more often complex, interactive, event-based processes. APR results on Defects4J do not reflect the results on everyday systems. Another problem, especially when using large language model-based AI for APR, is that the sources both for the original and repaired code for the examples in Defects4J are publicly available and have most likely been used to train the LLMs. Since new systems and new bugs do not satisfy these criteria, evaluation results might not reflect using APR on real systems.

A related issue is that test cases are difficult to create for event-based systems where events can be triggered by user interaction, by other processes, by timers, or by real-world actions. Bugs in these systems can involve the interactions between various events, timing issues, and the unexpected behavior of other processes and users. Creating test cases for this kind of problem is difficult, if it is even possible. Moreover, the number of possible interactions and situations to consider grows exponentially, as would the number of necessary test cases. Similar issues arise in finding and fixing bugs related to graphical input and output. Again, because such applications and bugs are precluded from test-case-based evaluation, the evaluation results might be biased.

The next problem is in the process of creating a test case, much of the debugging has already been done. Koyuncu et al. [15] noted that bugs are often reported without an available test suite being able to reveal them. Over 90% of the test cases in Defects4J were introduced only after the bug was identified. Creating a test case based on a bug report or problem observed at the system level involves doing considerable fault localization and debugging in order to identify a particular routine and and its arguments that illustrate the problem and can be called by a test case. Thus, using test cases to drive APR oversimplifies the problem since the bugs that are addressed have already been simplified and analyzed.

The next issue is that test cases are too coarse a measure for validating a patch. The assumption is that test cases embody the semantics of a program. This is what causes the overfitting problems that plagued early bug repair efforts. While test cases were the best informal approach one could take for a long time, this is no longer true. Large language models (LLMs) let one use documentation, natural language, or even the code itself and can do a reasonable job of understanding what the code should do. The validation issue

APR 2026, Munich, Germany

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

today is more of a problem in those systems that try to find a complex repair involving multiple changes incrementally and thus need to know whether an initial patch is a step in the right direction. It is unlikely that there are multiple test cases where the first part of a multi-part fix actually changes the number of tests passed for a complex bug [20, 26].

The next issue is that test-based APR has typically assumed spectra-based fault localization techniques, where coverage differences between passing and failing tests are used to identify suspicious lines. These techniques are only moderately successful [1, 23]. This is one of the reasons why evaluations of APR systems often use the perfect fault localization assumption. However, more information than the fact that a test failed is actually available for APR, notably why the test failed, for example, if it was due to an exception, what caused the exception and what led to that cause. Using this additional information can yield more accurate fault localization and hence better program repair. Moreover, without a substantial test suite, spectra-based fault localization is not available to APR techniques. The availability of large test suites has resulted in most APR techniques ignoring the fault localization problem which should be considered an integral part of the process.

The final issue is that test cases can be slow. Running a single test case can take anywhere from a fraction of a second to minutes. Running a single test case multiple times to validate patches takes minutes. Running a whole test suite can take hours. The result of this is that APR has typically been viewed as a batch process where APR is fully responsible for finding the correct fix with no developer supervision. However, choosing the correct patch can be difficult and requires a solid understanding of the overall system semantics, both current and future. This is something the developer knows but the APR system does not. Considering interactive APR makes sense, but is often precluded when using test cases.

We as a community need to do APR without relying on an extensive test suite or, in many cases, even a single test case. At the same time, we want APR to handle the wide range of problems that test cases can identify as well as those that arise in real applications.

2 Desired Approach to APR

Practical APR should work with a variety of different types of applications and a variety of different types of bugs. It should work as an assistant to the developer.

APR should start by identifying a symptom that exemplifies a problem that needs to be fixed. This can be a failing test case (with or without a test suite), but it can just as easily be a bug report, the errors found during static analysis, error messages from a log file, or the developer noting a problem such as an exception during debugging. The problem to be repaired is then defined by this symptom and a description of the desired program behavior.

APR should ideally find the "correct" repair for this problem. This is not always possible or practical because the correct repair can be ambiguous, can depend on the complete and future semantics of the application, and can require a substantial rewrite of the application. Moreover, APR systems are known to overfit patches to test cases or to hallucinate fixes or problems when using LLMs. APR should explain what causes the symptom, identify where in the code the problem lies, and suggest possible patches, including complex fixes that might require multiple changes in various parts of the application. APR should also validate the fixes as much as possible, providing confidence levels where appropriate. Unless the patch is an obvious and simple fix, the developer should vet the patches and accept what they consider a correct one.

This should be done in the context where the problem was identified. If the problem was identified while debugging, this should be done in the context of the debugger and the developer should be able to fix and continue. If the problem was identified during static analysis, then the failing static analysis should be rerun. If the problem was identified while testing, the program should be retested. Moreover, this should be done in a timely fashion, especially if the developer is involved in selecting the appropriate patches.

3 APR Without Test Cases

Doing all this without a test suite or even a test case can be difficult. APR first needs to define the expected behavior of the application, identifying the symptom and what should have occurred. When testing this is just the failing test case and the fact that the test should have passed. Next, APR needs to identify which parts of the code might cause the problem. With a test suite, one can do spectra-based fault localization to identify particular lines. Then APR needs to suggest patches for these lines or for their containing methods using one of the myriad of methods that have been proposed and used. Complications arise when these patches are complex requiring multiple changes possibly in multiple methods. Finally, APR needs to validate the patch, showing that it removed the symptom and caused the program to behave correctly. With a test case, this can be the test case passing without any other tests that used to pass failing.

Thus, test cases are tightly integrated into existing APR techniques and performing APR without a test case is challenging and difficult. Nevertheless, there have been several approaches that attempt to do this, albeit in specialized domains.

A group of test-free techniques focus on addressing security vulnerabilities using methods such as constraint-based code synthesis [8], neural machine translation [5, 7], invariant inference [29], and safety property guidance [9]. Another group targets GitHub issues. Jimenez et al. [11] used LLMs to generate patches based on the issue description and the project code. SWE-agent [27] and AutoCodeRover [30] use LLMs as agents to analyze the issue, search for code, and propose patches. CrossFix [24] searches for similar GitHub issues or bugs and leverages their fixes for patch generation. ExtractFix [8] extracts a constraint encoding the expected program behavior and synthesizes patches from that.

Getafix [2], Phoenix [3], FootPatch [25], and SymlogRepair [18] are repair techniques that rely on static analyzers for bug detection and repair validation. Addressing the problem of providing assistance from within the debugger has been more recent [19].

LLMs for programming have improved significantly so that today's models are capable of doing a lot of the work of APR. Agentic LLMs can do bug diagnosis, fault localization, and patch generation for simple examples. For example, **ChatDBG** [17] and **Cursor** [6] integrate an LLM with the debugger so that the LLM controls the debugger through agents. This approach does not provide a unified integration of extensive static-flow, slicing, and repair validation

for various debugging contexts, limit the types and complexity of bugs that can be handled [13].

4 Our Approach and its Limitations

Our efforts at non-test-based APR, originally ROSE [22] and now DIAD [21], try to be more general and to handle a wider variety of bugs. In these systems we make the assumption that the developer is working in the debugger and the program has stopped. This might be at an exception, at a breakpoint, say in defensive code, or after a sequence of debugging actions. We try to be general within this context by handling a variety of symptoms. Although somewhat limited, identifying a problem while debugging is a fairly common occurrence during development.

We first need to identify the symptom of the problem. In ROSE we ask the programmer to define it from a fixed set of types. DIAD does a detailed analysis of the stopping location and the run time environment and attempts to automatically define an appropriate symptom, letting the developer change this if necessary. DIAD also takes a broader approach to symptoms by using LLMs to do the underlying analysis. The developer can describe an arbitrary symptom, but is then asked to optionally identify variables that affect that symptom to facilitate fault localization and bug analysis.

Next, ROSE and DIAD identify potential source locations that affect the symptom and hence might be candidates for patching. This starts from a set of variables or run time values based on the symptom, uses a flow analysis tool that is part of the IDE, and does a backward slice to identify the set of lines and methods that affect these values. The result is a set of all locations that might contain an error. This is not perfect: the set can become quite large and is then restricted by PDG distance; and the resultant set is not necessarily accurate for errors of omission. The flow analysis needs to deal with language issues including reflection and native methods and takes some short cuts for performance. To allow for better accuracy, the developer can customize the analysis for the application and the libraries used.

The systems then use a live-programming extension built into the IDE to get an execution trace for a relevant portion of the current call stack. This can fail for several reasons. First, it requires that the execution can be restarted and will still match the current debugger state. This can require resetting values that have changed, for example, removing an element from a DONE set so it will be considered again in the simulated run. ROSE and DIAD handle this using a set of heuristics. Second, it is difficult for live programming to handle items such as native methods, database access, and input from sockets or previously opened files. The trace is used to identify the potential fault locations that are actually executed and to provide additional information for later analysis.

ROSE then performs a broad search, looking for possible patches at each identified location using a combination of patch generators based on previous APR techniques. It attempts to validate each patch using the live programming extensions. It first checks that the symptom disappeared, for example that a previously thrown exception is not thrown with the patched code. It also checks if execution of the modified program runs to completion without other errors. It assigns a priority score to each patch.

DIAD asks an LLM to explain the symptom, identifying its root cause. It used agents to provide the LLM with access to the backward slice results, to the execution trace, to semantic and syntactic program information, and to the source code.

Finally, both ROSE and DIAD interact with the developer to choose the most appropriate patch. ROSE shows the set of patches found along with their priority ratings. It lets the developer see the actual changes for a patch before having the system actually change the code, allowing fix and continue if the system allows. DIAD shows the explanation to the developer and lets them ask the system to try the explanation again, possibly with additional information. DIAD can also show the patch suggested by the LLM to the developer and let the developer edit the result. It can change the original source code once the developer approves.

We note that ROSE and the original version of DIAD only address one way of identifying a problem to fix: defining a symptom while actively debugging. There are several other ways in which a software issue can be identified. This includes test cases (which if they fail during debugging are essentially a type of problem DIAD can handle); stack traces that are included either in a bug report or in a log file; bug reports noting incorrect output; bug reports noting an incorrect display or other user interface issue; trace output that identifies a potential problem, for example, from defensive code; or semantic analysis that identifies potential security or system problems. We are extending DIAD to handle debugging stack traces, but the rest of these need to be handled in some other manner.

5 Challenges

The various techniques developed to perform APR without test cases demonstrate that such approaches are possible, even if they are not complete or are often impractical. Moreover, the success of modern LLMs in diagnosing and fixing problems without test cases shows that APR without test cases can be practical. This raises several challenges.

For our research, the challenges involve extending the applicability of our techniques by considering a broader range of symptoms, better fault localization, and more complex partial executions to understand what led to the problem. For example, we need to extend our approach to problems that arise outside the current call stack. We are currently working on extensions of DIAD to do APR directly from a stack trace, and to generate a test case that can duplicate an identified symptom.

For the APR community in general, we encourage researchers to stop assuming an extensive test suite for fault localization, patch validation, and experimental validation and to develop and incorporate other techniques. LLM-based techniques with agents specialized for debugging seem particularly promising. We also encourage APR techniques that are interactive rather than batch, techniques that involve the developer in the patching decisions. We also encourage researchers to address the other common methods in which potential bugs are identified.

More importantly, as a community, we need to develop evaluation criteria that can be applied fairly to a broad range of techniques and problems and do not assume a publicly available test suite. Without this, new approaches which target other bug situations will be discouraged since it will be much more difficult to demonstrate

their utility compared to existing systems and the evaluation results will continue to be biased and not necessarily reflective of the techniques being used on a user's applications. Limiting an evaluation of a technique that can handle a wide range of problems to only problems that are derived from test cases, and then comparing the results to previous results on the test suite does not show the advantages of the new technique. We are currently developing a test suite more suitable for evaluating DIAD, but this is not the broad-based suite that should be our target goal.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION)*. IEEE, 89–98. doi:10.1109/MUTATION.2007.13

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27. doi:10.1145/3360585

[3] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering*. 613–624. doi:10.1145/3338906.3338952

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting on the Foundations of Software Engineering*. 179–190. doi:10.1145/2786805.2786843

[5] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165. doi:10.1109/TSE.2022.3147265

[6] Cursor Debug 2026. *Debugging with Cursor.* https://cursor.com/for/debugging

[7] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 935–947. doi:10.1145/3540250.3549098

[8] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology* 30, 2 (2021), 1–27. doi:10.1145/3418461

[9] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–554. doi:10.1109/SP.2019.00071

[10] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. 2013. An unabridged source code dataset for research in software reuse. In *Proceedings of 10th Working Conference on Mining Software Repositories*. 339–342. doi:10.1109/MSR.2013.6624047

[11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023). doi:10.48550/arXiv.2310.06770

[12] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440. doi:10.1145/2610384.2628055

[13] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proc. ACM Softw. Eng.* 1, FSE, Article 64 (July 2024), 23 pages. doi:10.1145/3660771

[14] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. 2013. An empirical study of adoption of software testing in open source projects. In *Proceedings of 13th International Conference on Quality Software*. 103–112. doi:10.1109/QSIC.2013.57

[15] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering*. 314–325. doi:10.1145/3338906.3338935

[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A generic method for automatic software repair. *IEEE transactions on software engineering* (2011), 54–72. doi:10.1109/TSE.2011.104

[17] Kyla H Levin, Nicolas van Kempen, Emery D Berger, and Stephen N Freund. 2025. ChatDBG: Augmenting Debugging with Large Language Models. (2025). doi:10.1145/3729355

[18] Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. 2023. Program Repair Guided by Datalog-Defined Static Analysis. In *Proceedings of the 31st ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE) to appear*. doi:10.1145/3611643.3616363

[19] Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. 2024. An Empirical Study on LLM-based Agents for Automated Bug Fixing. *ArXiv* abs/2411.10213 (2024). doi:10.48550/arXiv.2411.10213

[20] Noor Nashid, Daniel Ding, Keheliya Gallaba, Ahmed E. Hassan, and Ali Mesbah. 2025. Characterizing Multi-Hunk Patches: Divergence, Proximity, and LLM Repair Challenges. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1629–1641. doi:10.1109/ASE63991.2025.00137

[21] Steven P. Reiss and Mohamed Salem. 2026. A Smart Assistant for Debugging. In *Proceedings of ASE 2026*. doi:10.1145/3832783.3834620

[22] Steven P. Reiss, Xuan Wei, Jiahao Yuan, and Qi Xin. 2025. ROSE: An IDE-Based Interactive Repair Framework for Debugging. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 112 (April 2025), 39 pages. doi:10.1145/3705306

[23] Manos Renieres and Steven P. Reiss. 2003. Fault localization with nearest neighbor queries. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 30–39. doi:10.1109/ASE.2003.1240292

[24] Shin Hwei Tan, Ziqiang Li, and Lu Yan. 2024. CrossFix: Resolution of GitHub issues via similar bugs recommendation. *Journal of Software: Evolution and Process* 36, 4 (2024), e2554. doi:10.1002/smr.2554

[25] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162. doi:10.1145/3180155.3180250

[26] Qi Xin, Haojun Wu, Jinran Tang, Xinyu Liu, Steven P. Reiss, and Jifeng Xuan. 2024. Detecting, Creating, Repairing, and Understanding Indivisible Multi-Hunk Bugs. *Proc. ACM Softw. Eng.* 1, FSE, Article 121 (July 2024), 24 pages. doi:10.1145/3660828

[27] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024). doi:10.48550/arXiv.2405.15793

[28] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang campaign, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2026. A survey on large language models for software engineering. *Science China Information Sciences* 69, 4 (2026), 141102. Originally published as arXiv preprint arXiv:2312.15223. doi:10.1007/s11432-025-4670-0

[29] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702. doi:10.1145/3533767.3534387

[30] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427* (2024). doi:10.1145/3650212.3680384