Combining Bug Detection and Test Case Generation

Martin Kellogg
University of Washington, USA
kelloggm@cs.washington.edu

ABSTRACT

Detecting bugs in software is an important software engineering activity. Static bug finding tools can assist in detecting bugs automatically, but they suffer from high false positive rates. Automatic test generation tools can generate test cases which can find bugs, but they suffer from an oracle problem. We present *N*-Prog, a hybrid of the two approaches. *N*-Prog iteratively presents the developer an interesting, real input/output pair. The developer either classifies it as a bug (when the output is incorrect) or adds it to the regression test suite (when the output is correct). *N*-Prog selects input/output pairs whose input produces different output on a mutated version of the program which passes the test suite of the original. In initial experiments, *N*-Prog detected bugs and rediscovered test cases that had been removed from a test suite.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;** *Maintaining software;*

Keywords

N-variant systems, mutational robustness, mutation, *N*-Prog

1. INTRODUCTION

Bugs are pervasive and expensive, and mature software projects ship with both known and unknown defects [1, 10]. Before fixing bugs, developers need evidence of their presence [23]—and acquiring this evidence earlier in the software lifecycle reduces each bug's cost [25]. To prevent defects in software shipped to customers, developers often use some combination of manual inspection, static analysis, and testing.

Each of these kinds of analysis has costs. Manual inspection is expensive and in modern practice is primarily used not to find defects, but rather for its other benefits [3]. Static analysis tools [2, 4] can detect many classes of defects but suffer from false positives—their warnings may or may not correspond to real defects in the code, and the outputs of such tools can be so large that they overwhelm users and lead to tool abandonment [7]. Testing can in theory detect any bug, but in practice test suites are often incomplete; writing tests manually takes significant developer effort, and automatic test generation tools [11, 19] suffer from an "oracle" problem—checking that outputs are correct requires that the tool know what the program under test should do [5].

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983970

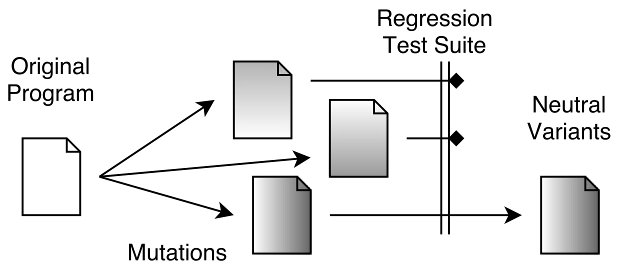

Figure 1: *N*-Prog's variant generation process. Mutation operators are applied to the original program to create candidate variants. The candidate variants are run against the existing test suite (the double line in the figure). Neutral variants are those that pass.

We present an initial look at a technique, *N*-Prog, that bridges the gap between static bug finding techniques and test suite generation by using the excess information and effort in each activity to complement the other. *N*-Prog presents its user with *alarms*, each of which either is a new test case, including the correct output, or indicates a bug in the program, along with some information that can be used to aid in fault localization. An alarm produced by *N*-Prog must be one of these two things, and the tool, by construction, therefore produces no false positives—in a sense, *N*-Prog replaces the "spurious warning" false positive of static analysis tools with useful new regression tests, each of which kills a mutant that the test suite could not differentiate from the original. As long as both bug detection and the generation of test cases are valuable activities, *N*-Prog provides value to its users with minimal overhead.

2. ALGORITHM

N-Prog combines random mutation (as in mutation testing [12] or automated program repair [15, 17, 21]) and *N*-variant systems. A traditional *N*-variant system implements several different *variants* of the program—ideally with independent failure modes—and runs them in parallel [8].

N-Prog replaces the semantics-preserving mutation operators of traditional *N*-variant systems with statement-level mutation operators: *N*-Prog can delete an existing statement or insert a statement it finds elsewhere in the program. Figure 1 shows *N*-Prog's variant

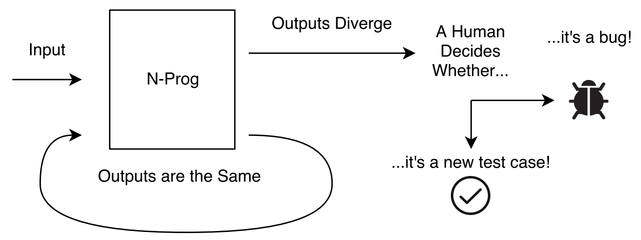

Figure 2: *N*-Prog's workflow. Only inputs that diverge between the original and at least one variant make it past *N*-Prog's filter to be seen by a human.

generation process. *N*-Prog first applies random mutations individually and tests them against the existing test suite. Those that fail a test are discarded, while those that pass—called *neutral*[1] mutants—are kept. From this list of neutral mutants, *N*-Prog creates higher-order mutants by combining individually neutral mutations. These higher-order mutants—if they stay neutral—are the variants that *N*-Prog deploys in its own internal *N*-variant system.

N-Prog then uses this *N*-variant system as a filter on an input source; Figure 2 shows a high–level view of how *N*-Prog is used. Any input source can work—random input from a tool like Randoop [19], data collected from users, or any other well-formed input source. If every variant exhibits the same behavior for a given input as the original program, then *N*-Prog ignores that input and moves onto the next; if there is at least one diverging variant, *N*-Prog will issue an alarm.

Once an alarm has been issued, there are two possible cases: either the original program is correct or the original program is incorrect. When it is incorrect, then *N*-Prog has exposed a bug in the implementation, and the developers have a candidate patch that causes the program to exhibit different behavior (the mutated variant). The existence of a patch, even if it is incorrect, has been shown to aid in debugging [24]. When the original program was correct, then *N*-Prog has generated a test case, and the original program serves as its own oracle. Once this test case is added to the suite being used to validate *N*-Prog's mutants, subsequent *N*-Prog variants will not alarm on that input. Notably, there is evidence that the generated test is of interest to the developers: in order to trigger an *N*-Prog alarm, it must kill a mutant that the original test suite could not kill—forcing that variant, originally uncovered by the test suite, to be covered.

For each alarm, the only thing the developers must do is examine the input/output pair (i.e., the given input and the original program's output) and determine whether the original program is behaving correctly. Usually, this requires much less effort than either writing a new test case from scratch or reproducing a bug—since either of those activities requires developers to examine input/output pairs, anyway.

3. EARLY RESULTS

To show that *N*-Prog can detect bugs, we ran *N*-Prog on programs with known defects, using as input the known (to the experimenter) buggy input. If *N*-Prog produces an alarm, we consider that a success; if *N*-Prog does not, a failure. We tested on the IntroClass suite of student-written programs [16]. It contains six small programs, each with real mistakes made by CS1 students.

[1]We follow Shulte et al. and use the biologically-inspired term *neutral* [22], but *test-equivalent* [13] and *sosie* [6] also appear in the literature.

Program	Scenarios	Alarms	Alarm%
`checksum`	61	48	79%
`digits`	199	170	85%
`grade`	252	219	87%
`median`	170	149	88%
`smallest`	117	113	97%
`syllables`	128	115	90%

Table 1: A demonstration of *N*-Prog's ability to detect bugs. Each scenario contains a held-out bug. The "alarms" column indicates in how many of the scenarios for a given program *N*-Prog can detect the bug.

Table 1 shows that *N*-Prog detected most of the bugs, usually within a few minutes of starting the run; the computational cost of running *N*-Prog is dominated by the cost of running the test suite of the system under test (since many variants need to be tested for neutrality).

We carried out several other experiments using *N*-Prog. These experiments are detailed in a technical report [14] and include a case study of *N*-Prog as it builds a test suite for a webserver—from about 140,000 test inputs, *N*-Prog selected a test suite of 25—as well as an evaluation of *N*-Prog's ability to find bugs on a collection of more realistic programs than those presented here.

4. RELATED WORK

Function. *N*-Prog combines bug detection and test case generation. Traditional bug detection tools, like FindBugs [2] or Coverity [7], use static analyses to detect bugs at compile time. A key weakness of these techniques is false positives; Coverity—a commercial tool—struggles to keep its false positive rates below 20–30%. While *N*-Prog may not be as effective at finding bugs as these specialized tools, it can in principle detect any bug that its mutation operators can touch, and does not suffer from false positives. *N*-Prog shares some goals with test case generation tools like Randoop [19] or EvoSuite [11], and can use such tools for input generation. EvoSuite uses mutation to generate oracles, but normally assumes that the program under test is correct.

Form. Mutation testing researchers use mutation to evaluate and augment test suites [12]. Mutation is also used in fault localization: tools like MUSE [18] or Metallaxis [20] use it to find the source code responsible for already-reproducible bugs. By contrast, *N*-Prog is used to find previously unknown bugs. Mutation-based generate-and-validate program repair tools, including GenProg [15], RSRepair [21], and Prophet [17] repair known defects. Schulte et al. [22] also used mutation to attempt to proactively repair defects before they had been detected. *N*-variant systems are used to provably defeat certain types of security vulnerabilities [9].

5. CONCLUSION

This paper presented *N*-Prog, a tool that combines bug detection with test case generation. *N*-Prog exploits weaknesses in each technique to augment the other: false positives become regression tests, and every time a human interacts with *N*-Prog, there is a positive outcome: either a bug is found or a useful, mutant killing test case—complete with oracle—is written.

6. ACKNOWLEDGMENTS

A special thanks to other project members Jamie Floyd, Stephanie Forrest, and Westley Weimer; and to Michael Ernst.

7. REFERENCES

[1] J. Anvik, L. Hiew, and G. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[3] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE Press, 2013.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages*, pages 1–3, 2002.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015.

[6] B. Baudry, S. Allier, and M. Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. *CoRR*, abs/1401.7635, 2014.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-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. *Communications of the ACM*, 53(2):66–75, 2010.

[8] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *International Conference on Fault Tolerant Computing*, pages 3–9, 1978.

[9] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *USENIX Security Symposium*, 2006.

[10] S. M. Donadelli, Y. C. Zhu, and P. C. Rigby. Organizational volatility and post-release defects: A replication case study using data from google chrome. In *Mining Software Repositories*, pages 391–395, May 2015.

[11] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Transactions on Software Engineering*, 38(2):278–292, 2012.

[12] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.

[13] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 315–326. ACM, 2014.

[14] M. Kellogg, B. Floyd, S. Forrest, and W. Weimer. Combining bug detection and test case generation. Technical report, University of Washington, September 2016.

[15] 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 *International Conference on Software Engineering*, 2012.

[16] C. Le Goues, N. Holtshulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. In *IEEE Transactions on Software Engineering*, 2015.

[17] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Principles of Programming Languages*, pages 298–312. ACM, 2016.

[18] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Software Testing, Verification and Validation (ICST)*, pages 153–162. IEEE, 2014.

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84. IEEE, 2007.

[20] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

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

[22] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[23] Y. Tian, D. Lo, X. Xia, and C. Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354–1383, 2015.

[24] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[25] L. Williamson. IBM Rational software analyzer: Beyond source code. In *Rational Software Developer Conference*, June 2008.