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 an experimental evaluation, *N*-Prog detects bugs and builds a test suite for a real webserver from an indicative workload.

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 [26]—and acquiring this evidence earlier in the software lifecycle reduces each bug's cost [28]. 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 [12, 22] suffer from an "oracle" problem—checking that outputs are correct requires that the tool know what the program under test should do [5].

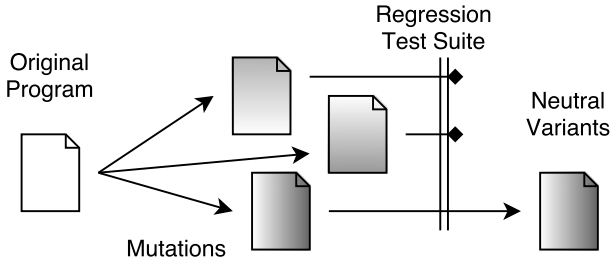

Figure 1: *N*-Prog's variant generation process. Mutation operators are applied to the original program to create candidate variants. The candidate variants are run against the existing test suite (the double line in the figure). Neutral variants are those that pass.

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 [14] or automated program repair [17, 20, 24]) 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

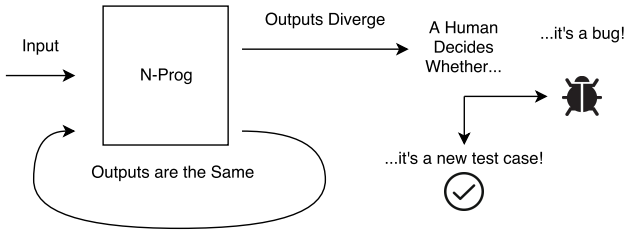

Figure 2: *N*-Prog's workflow. Only inputs that diverge between the original and at least one variant make it past *N*-Prog's filter to be seen by a human.

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 [22], 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 [27]. 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. EVALUATION

We evaluate *N*-Prog with a series of experiments that assess its different capabilities in isolation. Taken together, the experiments show that *N*-Prog can consistently provide value to the developer. Our experiments address the following four research questions:

- **RQ1:** How does *N*-Prog scale and does it apply to real-world workloads? (Section 3.1)

[1]We follow Shulte et al. and use the biologically-inspired term *neutral* [25], but *test-equivalent* [16] and *sosie* [6] also appear in the literature.

Figure 3: Alarms raised by *N*-Prog applied to a webserver with no test cases against an indicative workload of 138,266 requests (note log scale). Each alarm corresponds to a test case (input and oracle) discovered. The last alarm is raised at request 10,212.

- **RQ2:** Can *N*-Prog detect held-out defects in a variety of programs? (Section 3.2)
- **RQ3:** What types of defects can *N*-Prog detect? (Section 3.3)

Our evaluation features both real and toy programs, including some that are tested exhaustively, some with real-world test suites, and both those with real and with seeded defects. When checking for divergence, we always compare user-visible output (i.e. using `diff`). We try to select parameters that result in reasonable variant generation times. *N*-Prog is sensitive to both the number of variants deployed, and the number of mutations in each variant: higher values of either lead to more alarms but increase variant generation time. Increasing *N*-Prog's mutant generation budget (i.e. how long to spend generating mutants) slightly increases variant generation time and alarm rates, but the effect is so small is can be attributed to chance.

3.1 RQ1: How does *N*-Prog scale and does it apply to real-world workloads?

3.1.1 Benchmarks and Experimental Setup

This experiment measures how many alarms *N*-Prog raises on an indicative, non-bug-inducing webserver workload. The webserver starts with a single, simple test case, so each alarm during the experiment corresponds to a new test case that should be added to the test suite. We apply *N*-Prog to `lighttpd 1.4.17`, a Web 2.0 webserver with a historical workload of 138,226 benign HTTP requests spanning 12,743 distinct client IP addresses [19, Sec. 6.2]. *N*-Prog is configured with 8 variants of 30 edits each. These are subjected to the indicative workload. Each time *N*-Prog alarms, the input is added to the test suite, with the original output as the oracle, and the eight variants are regenerated. Since this experiment begins with a nearly empty test suite, we expect many alarms initially, as a test suite is generated to cover indicative behavior [14]. Since developer concerns over high alarm rates are an impediment for static analyses [15], ideally *N*-Prog will alarm only while building a good test suite.

3.1.2 Results

Figure 3 shows the results. *N*-Prog scales well, regenerating the eight variants only 25 times over the course of 138,226 requests. Since each alarm represents a human-time cost to determine if the alarm represents a test case or a defect as well as a CPU time cost

Program	LOC	Tests	Scen.	Alarm%
print_tokens*	472	4,140	7	29%
print_tokens2*	399	4,115	10	40%
replace*	512	5,542	31	32%
schedule*	292	2,650	9	11%
schedule2*	301	2,710	10	30%
tcas*	141	1,608	41	49%
tot_info*	440	1,052	23	30%
potion	15K	220	15	27%
gzip	491K	12	5	80%
php	1,046K	8,471	62	21%
Total	1,593K	30,070	213	32%

Table 1: Bug detection: Empirical results evaluating *N*-Prog's ability to detect bugs. Each scenario contains one held-out bug. The "alarm%" column reports in what percentage of the scenarios the held-out bug was detected by *N*-Prog for a given program. Programs labeled with a * are from the Siemens suite.

Defect Category	*N*-Prog
Incorrect Behavior or Output	6/30
Security Vulnerability	1/4
Missing Functionality	1/11
Missing Input Validation	2/3
Spurious Warning	1/2
URL Parsing Error	1/1
File I/O Error	1/1
Fatal Error	2/9
Segfault	1/8
Bounds Checking	0/1
Memory Leak	0/1
Total	17/67

Table 2: Number of bugs detected by *N*-Prog from the ManyBugs programs `gzip` and `php`, broken down by type. Note that bugs classified as "security vulnerability" are also included in another row by cause.

(to regenerate variants), minimizing this number is important. In this experiment all inputs are known to be benign, so we do not explicitly measure the human judgment cost.

The alarms occurred in the early part of the run rather than being evenly distributed. The last alarm occurs after only 7% of the input has been considered, which is relevant to the real-world experience of using the tool—developers will not be bothered by many redundant alarms once a good test suite exists. The experiment shows that as test cases are added to the test suite over time, *N*-Prog alarms only on novel inputs that exercise previously untested functionality.

3.2 RQ2: Can *N*-Prog detect held-out defects in a variety of programs?

3.2.1 Benchmarks and Experimental Setup

This experiment measures the percentage of buggy inputs (i.e. inputs we, the experimenters, know will trigger incorrect behavior) that *N*-Prog flags as interesting and brings to the attention of developers. The experiment considers 213 *scenarios*—versions of programs with a known bug and an input that triggers the bug. For each scenario, *N*-Prog generates an *N*-variant system, using the program's regression suite (without the buggy input). The buggy input is then used as the input stream; if *N*-Prog alarms, then *N*-Prog has detected the bug. If not, we count it as a failure. *N*-Prog is configured with 25 variants of 30 mutations each.

The benchmark set was selected from previously published work and includes both toy and real programs. The Siemens microbenchmark programs (marked with a * in Table 1) are small C programs that are exhaustively tested—each has at least 30 test cases covering every line of source code [13]. The seven Siemens programs have 131 buggy scenarios. We also included 15 scenarios of `potion`, an interpreter for a toy language [25]. Both the Siemens programs and `potion` contain artificial, seeded defects. To test *N*-Prog on more natural bugs, we included two programs from the ManyBugs benchmark suite, which are large, open-source programs with reported and repaired real bugs and developer-supplied test suites. We selected 67 scenarios from the `gzip` and `php` benchmark programs for which we were particularly confident of the test suites.

3.2.2 Results

The results appear in Table 1. Of the 213 buggy scenarios, *N*-Prog detected 68 (32%). *N*-Prog detected the bug in at least one scenario for each of the subject programs, and there were only two outliers among the ten programs considered: `gzip` and `schedule`. On `gzip`, *N*-Prog was much more successful than average, while on `schedule` it was much less successful[2]. The 32% detection rate is conservative, because if *N*-Prog has larger budgets (i.e. more variants in each system and more mutations per variant) it could be more effective (but more expensive to run). However, *N*-Prog's ability to detect about a third of defects is encouraging: if even a third of general software engineering defects can be detected automatically, while improving the subject program's test suite, a lot of developer effort can be saved.

3.3 RQ3: What types of defects can *N*-Prog detect?

Because *N*-Prog produces neutral variants, rather than equivalent mutants [14], it can theoretically produce divergence for arbitrary defects. We investigate what kinds of defects *N*-Prog detects in practice. The ManyBugs programs (`gzip` and `php`) contain real, historical defects, ranging from erroneous warning messages to security issues. And, these scenarios classify of each bug [18]. So, we examined the classifications of the scenarios from the ManyBugs suite in detail. Most of these defects were characterized as involving missing, extraneous or incorrect functionality.

Table 2 shows how many of each category of defect *N*-Prog was able to detect in the `gzip` and `php` scenarios. Defects include at least one (the "Security Vulnerability" in `php`) marked "security critical." *N*-Prog can detect defects across a broad range of defect categories, demonstrating its generality. In principle, *N*-Prog can detect any defect that a change introduced by its mutation operators can impact—allowing it to apply to general software engineering defects, and not just defects of a particular type.

Three of the detected defects were unusual: in each, an extraneous piece of information unique to the execution is printed to the output. For example, one bug in `php` (in the "Incorrect Behavior or Output" row of Table 2) involves printing the value of a memory address, instead of printing the contents of a data structure. Such defects could be detected by any neutral variant in an *N*-variant system.

These results show that *N*-Prog can detect a wide variety of defect classes, including both security vulnerabilities and typical software engineering problems such as incorrect output.

[2]We suspect that *N*-Prog is so successful on `gzip` because `gzip`'s regression suite is weak and `gzip`'s output is highly sensitive to the kind of changes *N*-Prog makes.

3.4 Threats to Validity

There are some threats to the validity of these experiments. First, the benchmarks may not be indicative, threatening the generality of our results. We mitigate this threat by selecting benchmarks from previously published projects and by choosing programs with different characteristics: sizes range from about 100 lines of code to more than a million; strong and weak test suites; and real and seeded bugs. Every benchmark is written in C—the *N*-Prog prototype only handles C, and programs written in other languages may have different behavior. A threat to construct validity (cf. [11, Sec. 2]) is the use of test cases as a proxy for indicative workloads or developer inspection. We address this directly in Section 3.1, using an historical indicative workload of 138,226 HTTP requests.

3.5 Verifiability

We have tried to make all of our experiments repeatable. Our prototype implementation of *N*-Prog is open-source[3]. The benchmarks are also available online[4], with instructions to reproduce each experiment.

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 [22] or EvoSuite [12], 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 [14]. Mutation is also used in fault localization: tools like MUSE [21] or Metallaxis [23] 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 [17], RSRepair [24], and Prophet [20] repair known defects. Schulte et al. [25] 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.

[3]https://github.com/kelloggm/nprog-code

[4]http://dijkstra.cs.virginia.edu/genprog/resources/nprog/icse-17-paper/

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] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research - an initial survey. In *Software Engineering and Knowledge Engineering Conference*, pages 374–379, Redwood City, San Fransisco Bay, CA, USA, 2010.

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

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200, 1994.

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

[15] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[16] 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.

[17] 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.

[18] 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.

[19] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

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

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