SHF: Small: Feedback-Driven Mutation Testing for Any Language

1 Overview and Objectives

1.1 Problem Statement

The core problem this project aims to address is making the use of program mutants practical in non-research settings, in a way that meets developers' or test engineers' needs; that is, making it possible for someone creating or enhancing a test suite, or developing code and test suite in tandem, to (1) use "just enough" mutation testing for their needs, maximizing benefit gained in exchange for work performed, (2) easily use custom mutation operators that target their specific software development task, and (3) work in any programming language without worrying about the quality of tool support provided for mutation testing. More generally, this project aims to make use of the insights of Test-Driven-Development (TDD), and proposes using mutation testing to move beyond a paradigm where developers build a series of tests narrowly tailored to steps in development, and use Mutation-Driven-Development (MDD) to build automated test generators or verification harnesses that handle not only anticpated problems imagined during development, but problems not anticipated by human insight, discovered using mutation-based analysis. In addition to traditional manual testing, our approach targets both highly-general property-driven testing and even full formal verification of software components, in order to be practical in the future, where software systems will often be so safety- or mission- critical that even "good" manual testing is not an acceptable approach to ensuring correctness, security, and reliability.

1.1.1 PI Qualifications

PI Groce ...

1.2 Intellectual Merit

2 Background and Preliminary Research

2.1 Furthest Point First and Fuzzer Taming

An unkilled mutant is, conceptually, very similar to a failing test. It presents information of possible relevance to a developer or test engineer. The mutant or test *may* indicate the presence of a previously unknown fault that needs to be fixed, either in the SUT or in the test suite/test generator. It may indicate the presence of a previously unknown fault of less importance. It may also indicate an even less interesting result: an equivalent mutant or a failure of an inherently flaky test. Or, in many cases, an unkilled mutant or failing test may contain information that is either important or unimportant, but is uninteresting because *it duplicates information already presented for understanding.* While examining an equivalent mutant is not always useless (e.g., it may indicate an opportunity for refactoring or improving the efficiency of code [36, 30]), examining a mutant that is equivalent to or extremely similar to an already-understood mutant is almost never worthwhile — even if the original mutant provided important, actionable information. That information has already been incorporated into the development or testing process.

Fuzzer taming [10] was a solution we proposed to the problem of triaging test failures in automated test generation. In compiler testing and other fuzzing applications, a core usability issue is that tools tend to produce very large numbers of failing tests for a much smaller number of distinct bugs. Finding the set of distinct bugs, and identifying important bugs that need to be fixed immediately is difficult, because the important bugs may be represented by only one or two failing tests in a set of thousands of failing tests, most of which are duplicates.

3 Research Plan

3.1 Core Research Questions

3.1.1 Research Questions Related to Feedback-Driven Mutation

1. How can we form a generalized, language-agnostic representation of and distance metric for program mutants for use in FPF?
2. How can we incorporate feedback from users into the representation and distance metric?
3. How can we set budgets for automated test generation and timeouts for verification efforts to quickly estimate whether a mutant is killable?
4. How can we most effectively use already generated killing tests and counterexamples to prune mutants?
5. Is distance-based clustering plus timing information useful for quickly eliminating killable mutants similar to already-killed mutants? How does this relate to Predictive Mutation Testing (PMT)?
6. How can we balance FPF-based selection of mutants for novelty with predictions of mutant equivalence, outcome, dominance, and productivity?
7. How can we identify outliers in otherwise similar groups of mutants, and is such identification useful?
8. Can we predict whether a mutant's unkillability is due to poor test generation or due to oracle weakness?

3.1.2 Research Questions Related to Any-Language Mutation

1. How can we maximize the efficiency and usability of a fundamentally language-agnostic regular-expression-based approach to mutant generation?
2. How can we extend the language of regular expressions to allow for language-agnostic definition of mutation operators that require more parsing-like analysis of code structure, without compromising the usability and simplicity of the approach?
3. Is it possible to perform on-the-fly mutant generation for very large projects, and reconcile this approach with FPF (e.g., generate new mutants with, possibly approximate, desired distances from already evaluated mutants)?

3.2 Work and Evaluation Plans

3.3 Evaluation Plan

4 Related Work

There is a vast body of work on mutation testing or analysis. Mathur attributes [47] the original idea for mutation analysis to a term paper by Richard Lipton in 1970. Foundational assumptions and theory were first proposed by DeMillo et al. [12], and the approach was first implemented by Budd et al. [9] in 1980. Mutation analysis as a theory relies on two fundamental assumptions — *the competent programmer hypothesis*, and *the coupling effect*, both of which have been widely studied [64, 65, 19, 54, 55, 42, 19, 16]. In this project, we are more interested in the practical effectiveness of mutation testing than in theoretical justifications.

It has long been argued [8] that mutation analysis is *stronger* than other coverage measures. The subsumption of multiple coverage measures by mutation analysis, including all the basic coverage measures [51] was shown by Offutt [57], and data flow subsumption was demonstrated by Mathur [49]. Daran et al. [11] found that mutation analysis produces faults that are similar to actual faults in terms of the error traces produced. Andrews et al. [5, 6] found that ease of detection of mutants was similar to that of real faults when compared to manually generated faults (in that manually generated faults were harder to find). Recent research by Just et al. [40] using 357 real faults showed that in 75% of cases, mutation score and test case effectiveness improved together, which is a strong relationship compared to the same coupling for coverage (46%). More recently, Papadakis et. al [59] showed in a large scale study that while there is a real relationship between mutation, it is in some sense weak; indeed they summarize their work by stating that "mutants provide good guidance for improving the fault detection of test suites, but their correlation with fault detection [is] weak," which is a foundational assumption of this proposal.

Cost of execution is often [39] considered to be the most problematic aspect of practical mutation testing. Numerous approaches exist, that seek to reduce the cost of mutation analysis. Offutt and Untch [56] categorize these, as: do *fewer*, do *smarter*, and do *faster* approaches. Operator selection, mutant sampling, and mutant clustering fall under *do fewer* — approaches that seek to reduce the number of mutants evaluated. The *do smarter* approaches seek to reduce the time taken for the entire mutation analysis by intelligently managing the various phases. Similarly, *do faster* approaches seek to reduce the time taken for evaluation of a single mutant, and include mutant schema generation, code patching, and other methods.

The *do fewer* approaches, especially simple random sampling, debuted with the initial research in mutation analysis [8, 1], where it was noticed that even a 10% random sample of mutants can on average be almost as effective (99%) as the complete set of mutants.. Sampling was further investigated by Mathur [46], Wong et al. [66, 67], and Offutt et al. [58].

Determining relative merits of selective mutation strategies such as operator selection and random sampling has long been an active field of research [67, 50, 73] Skew in fault representativeness among mutants was initially noticed by Budd et al. [8] who found that particular types of mutants are representative for particular kinds of faults. Constrained mutation was pioneered by Mathur [46, 48] and was further investigated by Wong et al. [68]. An extension of this approach called *n*-selection was suggested by Offutt et al. [58] where the most numerous mutation operators were removed one at a time. A set of guidelines for operator selection was identified and evaluated by Barbosa et al.[7]. Namin et al. [52, 53] formulated the concept of *sufficient mutation operators*, and Untch [63] even proposed simply using statement deletion as "the" mutation operator. Deng et al. [13] extended the deletion operator for diverse language elements, and obtained an effectiveness of 92% while reducing the number of mutants by 80%.

The subsumption of individual mutants and mutation operators is also an active area of research [17, 61, 45]. Higher order mutants (HOM) aim to improve the quality of mutants by combining simpler mutants into more complex mutants. Jia et al. [38, 37], found that the number of mutants can be reduced by 50% by making use of subsumption of simpler mutants by higher order mutants. Mutation clustering[14, 62, 35] is another *do-fewer* approach where similar mutants are identified based on various properties.

There has been extensive work on comparison of mutation reduction strategies [73, 72]. Zhang et al. [70] investigated the scalability of selective mutation by considering how well a randomly sampled set of mutants represent the original population. In our own previous work [18] we showed that there is an upper bound on the improvement in *mean effectiveness* that is possible using even an ideal mutation reduction strategy using post-hoc oracular knowledge of mutant kills. We later extended that result by evaluating the actual improvement achieved by extant mutation reduction strategies, when they do not unrealistically have access to the mutant kills achieved [20]. Recent work on Predictive Mutation Testing (PMT) [71] applies machine learning to build a model that can predict mutation results without actually running mutation testing, a novel and promising *do-smarter* approach.

4.1 Practical Mutation Analysis

The above work largely focuses on computing or at least estimating the total mutation score of a test suite, efficiently. The assumption is that mutation testing is meant to be, like a coverage metric, a kind of "evaluation" of a test suite, a number used to say "this is a good test suite" or at least "this is a better test suite than that test suite." While important for some real-world purposes (evaluating QA efforts) and certainly for software testing research, that is not the focus of this proposal. Instead, we consider the problem of presenting unkilled mutations to a developer or test engineer in a way that facilitates the improvement of a test suite and the detection of faults. This is inspired by our previous work on using mutation to find defects in formal verification and automated test generation efforts [27, 30, 2].

A second thrust of our efforts in this paper is to simplify the development, maintenance, and (especially) extension of mutation testing tools by using an extension of regular expressions to define mutation operators, and separating the generation of mutants from language or build-environment specific techniques for pruning invalid mutants. This aspect of the proposal primarily builds on our own initial work on the topic of regular-expression-based mutant generation [31] and Holzmann's work on lightweight textual code analysis with Cobra [34].

The single most relevant work by others to our aims is to be found in a report on actual techniques in use at Google for applying mutation testing to real-world projects [60]. Their approach also uses a notion of feedback, but this is manually handled and based on using a classification scheme to heuristically throw out "arid" (likely not to be actionable) mutants; due to the size of Google's code base and the integration of their approach in Google's code review process, there is also a decision to only allow one mutation (initially randomly decided) per line of code. While the underlying motivation, of giving developers actionable information rather than computing a mutation score, is similar, and the idea of using some form of "feedback" is common, our approach targets the individual developing, testing, or verifying a particular software element (either a small project, or a component of a project), and assumes an iterative process, where developers consider one unkilled mutant at a time. The Google approach does not attempt to prioritize between the unkilled mutants it surfaces, or support custom mutation operators, or learn an individual testing effort's characteristics. It is, instead, for obvious reasons, more an attempt to select mutants in an industrial scale code review setting than an effort to propose a new way to construct or enhance test suites; e.g., it only even proposes mutants of code in a diff with a previous version of the code, which makes it completely unusable in after-the-fact testing and verification efforts that do not have code changes. Further, it uses Google-wide coding conventions and developer suggestions to fix heuristics such as "avoid mutation of logging statements" rather than trying to learn (with human assistance) such heuristics for projects that may vary widely in language and coding style. We suspect that our ideas may be useful in generalizing or enhancing an effort such as Google's, however, and share the focus on mutants as tools for focusing developer/tester attention and producing action on the part of humans, not computing a mutation score. Indeed, the report itself allows that the current approach does not scale, since it requires extensive manual support for each heuristic and language, a problem this proposal aims to directly address.

More broadly, a paper by the authors of the Google report and a group of academic mutation testing researchers [36], uses the Google effort to propose a notion of productive and unproductive mutants. Their concepts are highly related to our goals, but again centered on a diff-focused, large-scale industrial setting, rather than an approach that, like TDD, may also be applied to smaller coding efforts in a more isolated setting, such as development of embedded software, where crowdsourcing is impractical. Another key difference is that, while their work used EvoSuite to enhance a test suite to kill additional mutants, the assumption was that most suite enhancement would be due to developers adding manual tests. Furthermore, we doubt that any mutant is either "productive" or "unproductive" in an absolute sense, but rather the value of a mutant also depends on previous mutants a developer has manually examined, and the results of that examination, and we embody this in an FPF-centric workflow.

5 Broader Impacts

Improving Software System Reliability:

Education and Outreach: The proposed research yields several opportunities for enhancing CS education, recruiting new CS majors, and retaining CS students, particularly members of underrepresented groups. PI Groce will work with the NAU Student ACM Chapter to present a series of "excursions in testing" that introduce automated testing to students, using feedback-driven mutation testing and mutation-driven-testing on real code, including code from media player libraries. The work of Guzdial [32] has shown that media computation is a potentially effective way to both recruit and retain female and under-represented minority students in computer science.

6 Results From Prior NSF Support

PI Groce has received support as PI or co-PI from three NSF grants. The most relevant and recent is “Diversity and Feedback in Random Testing for Systems Software” (CCF-1217824, $491,280, 9/2012–9/2017), a collaborative proposal with John Regehr at the University of Utah. **Intellectual Merit:** A key result from CCF-1217824 is the development of a strategy for creating “quick tests” [26], which won the Best Paper award at ICST 2014 for showing tests thus reduced can serve as effective regression tests or seeds for symbolic execution [22, 69]. Moreover, benefits do not depend on 100% preservation of a property [4]. Other results include an overview of the value of coverage in testing experiments [25] and exploration of how individual test features impact the coverage and fault detection statistics of random tests [24]. All of these results used mutation testing. **Broader Impacts:** CCF-1217824 has contributed to the discovery of previously unknown faults in multiple open-source and commercial software systems, including core compilers and system libraries. The development of the central swarm testing techniques has furthered many efforts to improve the quality of compilers, including LLVM and GCC, and to test core language tools in general [43, 41, 15, 44]. **Research Products:** Several publications resulted from this grant, including those cited above and numerous others [25, 10, 69, 26, 24, 3, 22, 33, 28, 4, 33, 21], along with three PhD theses. Source code [23, 29] is available on GitHub.

References

[1] Allen Troy Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[2] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. Applying mutation analysis on kernel test suites: an experience report. In *International Workshop on Mutation Analysis*, pages 110–115, March 2017.

[3] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 70–81, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi:10.1145/2931037.2931056. URL `http://doi.acm.org/10.1145/2931037.2931056`.

[4] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. Evaluating non-adequate test-case reduction. In *31st IEEE/ACM Conference on Automated Software Engineering*, pages 16–26, September 2016.

[5] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411. IEEE, 2005.

[6] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32 (8):608–624, 2006.

[7] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2): 113–136, 2001.

[8] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[9] Timothy Alan Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233. ACM, 1980.

[10] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[11] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 158–171. ACM, 1996. ISBN 0-89791-787-1.

[12] Richard A DeMillo and Richard J LiptonFrederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[13] Lin Deng, A Jefferson Offutt, and Nan Li. Empirical evaluation of the statement deletion mutation operator. In *IEEE 6th ICST*, Luxembourg, 2013.

[14] Anna Derezińska. Toward generalization of mutant clustering results in mutation testing. In *Soft Computing in Computer and Information Science*, pages 395–407. Springer, 2015.

[15] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 482–493. IEEE, 2015.

[16] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014.

[17] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. Measuring effectiveness of mutant sets. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2016.

[18] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. On the limits of mutation reduction strategies. In *International Conference on Software Engineering*. IEEE,

2016.

[19] Rahul Gopinath, , Carlos Jensen, and Alex Groce. The theory of composite faults. In *Software Testing, Verification and Validation (ICST), 2017 IEEE Eighth International Conference on*. IEEE, 2017.

[20] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 66(3):854–874, 2017.

[21] Alex Groce and Jervis Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[22] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*, . accepted for publication.

[23] Alex Groce, Yang Chen, John Regehr, Chaoqiang Zhang, and Amin Alipour. Swarmed versions of random testing tools. https://github.com/agroce/swarmed_tools, .

[24] Alex Groce, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Yang Chen, and John Regehr. Help, help, I'm being suppressed! the significance of suppressors in software testing. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 390–399, 2013.

[25] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 255–268, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1. doi:10.1145/2661136.2661157. URL `http://doi.acm.org/10.1145/2661136.2661157`.

[26] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 243–252. IEEE, 2014.

[27] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E McKenney. How verified is my code? falsification-driven verification. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 737–748. IEEE, 2015.

[28] Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.

[29] Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal, Josie Holmes, and Kevin Kellar. TSTL: the template scripting testing language. `https://github.com/agroce/tstl`, May 2015.

[30] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. How verified (or tested) is my code? falsification-driven verification and testing. *Automated Software Engineering Journal*, 2018. Accepted for publication.

[31] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 25–28, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5663-3. doi:10.1145/3183440.3183485. URL `http://doi.acm.org/10.1145/3183440.3183485`.

[32] Mark Guzdial. A media computation course for non-majors. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '03, pages 104–108, New York, NY, USA, 2003. ACM. ISBN 1-58113-672-2. doi:10.1145/961511.961542. URL `http://doi.acm.org/10.1145/961511.961542`.

[33] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, and James O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 2017. Accepted for publication.

[34] Gerard J. Holzmann. Cobra: Fast structural code checking (keynote). In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 1–8,

New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5077-8. doi:10.1145/3092282.3092313. URL `http://doi.acm.org/10.1145/3092282.3092313`.

[35] Shamaila Hussain. Mutation clustering. Master's thesis, Kings College London, Strand, London, 2008.

[36] Goran Petrović Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Proceedings of the International Workshop on Mutation Analysis (Mutation). IEEE Press, Piscataway, NJ, USA*, pages 47–53, 2018.

[37] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258. IEEE, 2008.

[38] Yue Jia and Mark Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51 (10):1379–1393, October 2009. ISSN 09505849.

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

[40] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 654–665, Hong Kong, China, 2014. ACM. ISBN 978-1-4503-3056-5.

[41] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: A language for property-based generators. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

[42] William B Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of systems and Software*, 83(12):2416–2430, 2010.

[43] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2014.

[44] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 327–337. ACM, 2015.

[45] Birgitta Lindström and András Márki. On redundant mutants and strong mutation. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2015.

[46] Aditya P Mathur. Performance, effectiveness, and reliability issues in software testing. In *Annual International Computer Software and Applications Conference, COMPSAC*, pages 604–605, 1991. doi:10.1109/CMPSAC.1991.170248.

[47] Aditya P Mathur. *Foundations of Software Testing*. Addison-Wesley, 2012.

[48] Aditya P Mathur and Weichen Eric Wong. Evaluation of the cost of alternate mutation strategies. In *Brazilian Symposium on Software Engineering (SBES)*, pages 320–335. Brazilian Computer Society, 1993.

[49] Aditya P Mathur and Weichen Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[50] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[51] Glenford J Myers. The art of software testing. *A Willy-Interscience Pub*, 1979.

[52] Akbar Siami Namin and James H Andrews. Finding sufficient mutation operators via variable reduction. In *Workshop on Mutation Analysis*, page 5, 2006.

[53] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*, pages 351–360. ACM, 2008.

[54] A Jefferson Offutt. The Coupling Effect : Fact or Fiction? *ACM SIGSOFT Software Engineering Notes*,

14(8):131–140, November 1989. ISSN 0163-5948. doi:10.1145/75309.75324.

[55] A Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[56] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[57] A Jefferson Offutt and Jeffrey M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, 1996.

[58] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*, pages 100–107. IEEE Computer Society Press, 1993.

[59] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, pages 537–548, 2018.

[60] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 163–171, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5659-6. doi:10.1145/3183519.3183521. URL `http://doi.acm.org/10.1145/3183519.3183521`.

[61] Donghwan Shin and Doo-Hwan Bae. A theoretical framework for understanding mutation-based testing methods. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2016.

[62] Joanna Strug and Barbara Strug. Machine learning approach in mutation testing. In *Testing Software and Systems*, pages 200–214. Springer, 2012.

[63] Roland H Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-421-8.

[64] K S How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, 2000. ISSN 1099-1689.

[65] K S How Tai Wah. An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2):119–161, 2003.

[66] Weichen Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, West Lafayette, IN, USA, 1993. UMI Order No. GAX94-20921.

[67] Weichen Eric Wong and Aditya P Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995. ISSN 0164-1212.

[68] Weichen Eric Wong, Marcio Eduardo Delamaro, José Carlos Maldonado, and Aditya P Mathur. Constrained mutation in c programs. In *Brazilian Symposium on Software Engineering (SBES)*, pages 439–452. Brazilian Computer Society, 1994.

[69] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 160–170, 2014.

[70] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. An empirical study on the scalability of selective mutation testing. In *International Symposium on Software Reliability Engineering*, pages 277–287, 2014.

[71] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *International Symposium on Software Testing and Analysis*, pages 342–353, 2016.

[72] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In *IEEE/ACM Automated Software Engineering*. ACM, 2013.

[73] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *International Conference on Software Engineering*, pages 435–444, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6.