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) work in any programming language without worrying about the quality of tool support provided for mutation testing, and without sacrificing the ease of understanding of source-based mutants, while easily adding custom mutation operators that target their specific software development task. 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 anticipated 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 Just Enough Mutation Testing: Feedback-Driven Mutation Testing

Most mutation testing work focuses on computing a mutation score or, at least, the set of unkilled (and perhaps non-equivalent) mutants. Using mutation testing tools involves running many tests on many modified versions of a software system, and, for larger projects or more expensive test suites, requires substantial computing resources, making reducing that need a major thrust of mutation testing research [67]. However, as useful as knowing the overall quality of a test suite may be, the most practical goal of mutation testing is to improve a test suite. For this purpose, an expensive-to-generate list of all unkilled mutants is not really what users need or want. A list of unkilled mutants contains uninteresting mutants (many, but not all, equivalent mutants), numerous redundant mutants (that can be killed by the same extension of the test suite, or rejected as uninteresting for the same reason), and a smaller number of actionable, representative mutants that are maximally effective in guiding improvement of a testing effort. Examining all unkilled mutants is only practical for formal verification efforts or very high-powered test suites and critical software systems that motivate such efforts. In our own work on using mutants to drive formal verification and automated testing [49, 53, 3] we note that examining surviving mutants was a time-consuming and unpleasant task, even in these settings. With a larger number of unkilled mutants, the problem becomes one very much like the bug triage or "fuzzer taming" problem in random testing/fuzzing [19, 117]: a user wants to quickly find mutants that indicate the most important "holes" in a testing or verification effort, and act on those most-critical gaps, possibly revealing faults in the System Under Test (SUT).

What a user really wants is a tool that presents a few very different, ranked, mutants, all likely to be of interest, and revises the presented mutants and their ranking based on actions taken by the user — adding tests, fixing faults, marking certain mutants as equivalent or uninteresting, and perhaps assigning a priority and severity to both killed or dismissed mutants and any remaining un-handled mutants. However, current mutation testing approaches make no real effort, with few exceptions [102, 16] to prioritize mutants, and none are based on a user-centered feedback loop, where the user and mutation testing framework interact to improve a test suite, automated test generator, or verification harness — and, of course, improve the SUT as well. To our knowledge, in fact, other than (arguably) some efforts to incorporate dominance results [99],

```
\+ ==> -            == ==> !=         (\D )(\d +)(\D ) ==> \1(\2+1)\3
\+ ==> *            == ==> <          (\D )(\d +)(\D ) ==> \1(\2-1)\3
\+ ==> /            == ==> >          (\D )(\d +)(\D ) ==> \g <1>0\3
".+" ==> ""         while ==> if      (^\s *)(\S +.*)\n ==> \1\2\n \1break;\n
```

Figure 1: Some universal mutation rules

no mutation testing approaches currently suggest any more sophisticated way to maximize the novelty of presented mutants than stratified sampling; stratified sampling does not really aim at semantic novelty, and can present many mutants from the same class together, if applied at the method level, even if those mutants are really quite similar in impact on the software system.

Problem: Develop highly automated methods and tools that allow the practical application of mutation testing in a feedback-driven way, where user and mutation testing framework cooperate to improve testing efforts, while minimizing user effort and maximizing the ability to quickly find the most important weaknesses of a test suite, automated test generation system, or formal verification effort.

1.1.2 Any-Language Mutation Testing

One ongoing limitation of mutation testing is that tools are often research projects, and eventually become unusable due to lack of support, even in mainstream languages such as Java and C [35]. This is because mutation tools that parse a language and guarantee generation of valid programs in the source language are complex, hard-to-maintain-and-extend systems; language complexity makes such a tool for C++, for example, an extremely daunting task. The most widely used mutation tool in the real world, as far as we are aware, is PIT [20], which targets Java bytecode. There are recent attempts to provide the same kind of support for other languages, especially C, by targeting LLVM IR [57].

The problem with bytecode-level mutation is that while an arguably excellent choice if the goal of mutation testing is to compute a score for a test suite, bytecode-level mutants are not nearly as suitable for presentation to users. A bytecode-level mutation may not have a source-level equivalent that is conceptually simple, especially if the bytecode has been optimized, and some obvious source-level mutations (such as statement deletion) are known to be difficult to precisely implement in bytecode. Moreover, targeting bytecode only helps with languages that compile to Java bytecode or LLVM IR, which leaves out Python, Ruby, Go, and numerous other popular languages, and certainly means that supporting project-specific Domain Specific Languages (DSLs) [27] is out of the question.

We recently proposed [54] an approach to mutant generation that does not attempt to parse source code, but simply defines mutation operators by a set of regular-expression-defined text transformations. These are organzied into a hierarchy, so that if a program is, e.g., written in Swift, the "universal" mutation operators that apply to all programming languages are first applied, then operators for "C-like" languages, and finally a set of Swift-specific rules are applied. Figure 1 shows some of the current set of "universal" rules applied to all languages. Adding a new language, even a custom DSL, or a new set of project-specific rules for an existing language, in this approach, simply requires writing a new rule file and defining where it lies in the language hierarchy. This approach is also attractive in our feedback-based setting, since the problem of generating "too many" mutants is irrelevant if only a small set of highly diverse and likely-actionable mutants is ever presented to the user, and a novelty-estimator helps a user stop examining new mutants when the payoff is likely to be low.

There are significant limitations to this approach, however. Because the source code is not parsed, and applies the regular expressions to lines of code, not larger blocks, the technique generates many mutants that are not valid programs, and cannot be compiled, or that are trivially equivalent because they, e.g., mutate "source code" in a large comment block. Integrating mutation generation with execution is currently supported, but extending it to new languages or build systems is hard for users, requiring writing considerable Python code or complex shell scripts. It is currently impossible to define mutation operators that apply to blocks of code rather than text within a single line, and standard regular expressions are not really suited to describing code constructs such as blocks, functions, classes, or structs. Natural formatting of, e.g., an s-expression in a LISP-family language can hide opportunities for mutation, such as switching argument orders.

Problem: Provide high-quality mutation generation support to be used in a maximally flexible but still efficient mutation testing framework that can be applied to (essentially) any language, such that adding project-specific novel mutation operators, or even adding support for a new language (e.g. a DSL used in a single project) is possible for non-expert users.

1.1.3 Mutation-Driven-Development

The primary focus of this project is to develop the algorithms and methods required for feedback-driven mutation testing, a process in which a user improves a verification or testing effort using a small number of well-selected mutants. However, the ideas of Test-Driven Development (TDD) [11, 64], which repeatedly turns requirements into specific test cases, then implements just enough functionality to pass the current tests (and assumes the previous implementation will not pass those tests), can be translated into a falsification-driven/mutation-driven form. A potential weakness (and, of course, an actual goal) of TDD is that the code will be narrowly tailored to the requirements, which produce the tests, which means that missing requirements will almost always be omitted both from the tests and the code. For "shall" type behaviors [61], this is not a key problem; missing "shall" requirements will likely be omitted in any case. But for security and safety, "shall not" requirements that are omitted can be deadly. Mutation-Driven-Development (MDD) in its simplest form would require an application of feedback-driven mutation to the test suite at each development step or at least at major milestones, to ensure that code not only does what the tests require, but that the tests also sufficiently constrain the code to capture at least many implicit shall-nots. Since such a process implemented by modifying TDD-driven tests would likely break the clean and appealing mapping between tests and requirements, and manual tests are inherently limited in effectiveness, for high-criticality systems, we suggest MDD should focus on augmenting TDD-driven tests with falsification-driven formal verification or at least automated testing. One way to do this would be to "elaborate" TDD-produced unit tests into parameterized unit tests [114, 113], perhaps using a tool like DeepState [30] for C/C++. In such a process, weakness exposed by feedback-driven mutation testing would normally be addressed by taking an existing unit test and generalizing some parameters and assertions to kill the relevant mutants, letting AFL [125], libFuzzer [108], or a symbolic execution tool [110, 111, 88] identify specific inputs. The focus of the MDD process would be on producing the test generator, test harness [38, 59], or parameterized version of unit test that allows an automated tool to quickly kill mutants covered by additional specification. More radically, MDD could be interpreted as a radical departure from normal TDD practice, where a single model checking or testing harness or parameterized unit test is iteratively enhanced with assertions and checks drawn from more requirements, always requiring the ability to kill (most) mutants of the current implementation that implements those requirements. This does not produce the large set of tests in TDD, but instead produces a single, monolithic, high-powered test generator or formal specification.

1.2 PI Qualifications

PI Groce has been a user of, and contributor to, mutation testing tools for many years. He combines a long research track record in software testing, including mutation testing, with actual experience testing critical software systems at NASA's Jet Propulsion Laboratory. PI Groce's long-running interest in improving the state-of-the-art in mutation testing dates from frustration in his efforts to apply mutation tools to the testing and verification effort for the Mars Science Laboratory's flight software, in particular to the file system [42, 43, 47]. This practical orientation informs recent work on using mutation testing in a falsification-driven approach to improving high-end verification and automated testing efforts [49, 53, 3]. PI Groce has extensive experience in developing mutation tools for new languages [76, 81, 54], including the first reliable tools for mutation of Haskell, Python, and Swift, as well as in user-facing (vs. researcher-oriented) automated software testing tools [59, 30].

1.3 Intellectual Merit

This work addresses core problems not limited to practical application of mutation testing to improve test efforts in a user-centered way, but generalizeable to fundamental issues in software engineering and program semantics. E.g., how can we represent program changes and (mostly statically) predict the similarity of their impact on program semantics, and which tests are likely to detect these changes. How can novelty of information presented to a user be effectively balanced with a-priori predictions of the utility of that information, where likely-high-utility data points may also be unfortunately similar to each other? How can user feedback be incorporated into such efforts without over-burdening human users? This project also considers connections raised by our preliminary work, concerning effective methodologies for program and testing/verification effort development. Can theoretical ideas about the nature of scientific discovery [103, 104, 73] be applied to such efforts? Is falsification by alternative hypotheses about the correctness of a system or power of a testing/verification effort translateable to an actionable, effective approach for building systems [49, 53]? The work on any-language mutation testing looks at syntactic patterns common to almost all programming languages, and relies on categorizing languages into families based on similarity, and how they share common meaningful syntactic changes that translate to interesting semantic changes.

2 Background and Preliminary Research

2.1 Falsification-Driven Verification and Testing

We recently proposed a novel approach to organizing and evaluating formal verification, automated testing, and, more generally, any testing efforts aiming at very-high-quality fault detection based on combining the insight's of Popper's falsification-based notion of scientific discovery [103, 104] with mutation based methods [49, 53, 3]. The heart of the idea is (1) the proposition that a surviving mutant that is not equivalent (or at least equivalent with respect to a given specification of correctness) *falsifies* the claim that a given formal verification or test effort captures the full notion of correctness for a sytem and (2) the proposal that refining a verification or test effort by repeated efforts at falsification is an effective method for ensuring the quality of verification and testing efforts. This line of work work resulted in the identification of multiple previously unknown faults in the Linux kernel's RCU [25, 55, 87] module and the `pyfakefs` Python mock file system [89], despite the existence of very-high-quality automated test generation efforts for these systems [86, 51]. We proposed a number of algorithms and methods for using unkilled mutants to guide model checking efforts, estimate needed random testing budgets and loop-bounds in bounded model checking [70, 12], and find bugs in testing and verification harnesses, rather than the code under test. At a high level, however, the core concept was simple: users should examine all unkilled mutants of a program, and for each mutant either understand why it is equivalent or uninteresting, or actually construct (with automated help) a way to kill it. In a sense, this simply harkens back to the earliest ideas about mutation testing, but with the addition of considerable automated support, at least for verification and automated testing efforts.

Unfortunately, the methods proposed were, while useful, limited in applicability. We simply assumed that the number of unkilled mutants to be examined was small, and focused on solving the problem of helping a developer or test engineer move from an unkilled mutant to an improvement of an already very-high-quality model-checking harness or automated test generator. Human attention does not scale to analyzing large numbers of unkilled mutants without further assistance in "triaging" the mutants, however. Our manual mutant-examination process, even aided by tools for handling individual mutants, was simply not feasible unless the number of unkilled mutants was relatively small, because the testing was already very high quality. Moreover, for a sufficiently large software system, even a very high quality verification or testing effort may fail to kill a large absolute number of mutants. Our approach simply provided no way to scale human efforts to such a needle-in-a-haystack setting.

This project aims to make the falsification-driven approach to verification and testing feasible for larger projects, or those with less effective testing or verification, and extremely easy for smaller projects with already-high-quality testing or verification. Moreover, because we no longer require small absolute numbers of unkilled mutants, we aim to extend the applicability of the approach to manual construction of tests to kill mutants, which was not in scope when extremely high kill-rates were required.

2.2 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 [63, 53]), 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 [19] was a solution we proposed to the problem of triaging test failures in automated test generation [117]. 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. We proposed that rather than highly imprecise clustering, which does not work well in practice, and handles outliers in a way that does not match the "power law" distribution bugs, an algorithm matching the goal of ranking maximally-different test failures highly was appropriate. Users do not (usually) care much about finding the group of all tests failing due to a fault, or the set of all mutants killable by the same extension to a test suite or generator, but about seeing *many very different test failures* or *many different unkilled mutants* quickly, to maximize the chance of discovering the most important faults or holes in a testing effort. The *furthest-point-first* (FPF) algorithm of Gonzalez [29] does precisely this. FPF, beginning with any randomly chosen test (or mutant, in our setting), always ranks next the point in a metric-defined space that has the *greatest distance from the previously ranked point to which it is closest.* That is, for each point (test or mutant) not yet presented to the user, FPF finds the closest among all already-ranked points, and associates each unranked point with the distance to that closest point. The unranked point with the largest such distance is then added to the ranking, and the process is repeated. FPF can be computed by a greedy algorithm, and is known to approximate novel-item discovery for an optimal clustering [29]. Our preliminary work on the fuzzer taming problem using FPF-based techniques [19, 58] can, we believe, be directly applied to the similar (but operationally quite different) problem of ranking unkilled mutants such that novel mutants are presented to a user. We discuss below, as part of our research plan (Section 3.1.1), the key *differences* between novelty ranking for mutants and the fuzzer taming problem.

2.3 Any-Language Regular-Expression Based Mutation

As discussed in the problem statement, we have released a functional, regular-expression-based mutant generator [54, 41], and demonstrated that it generated numbers of mutants and kill ratios for Java code

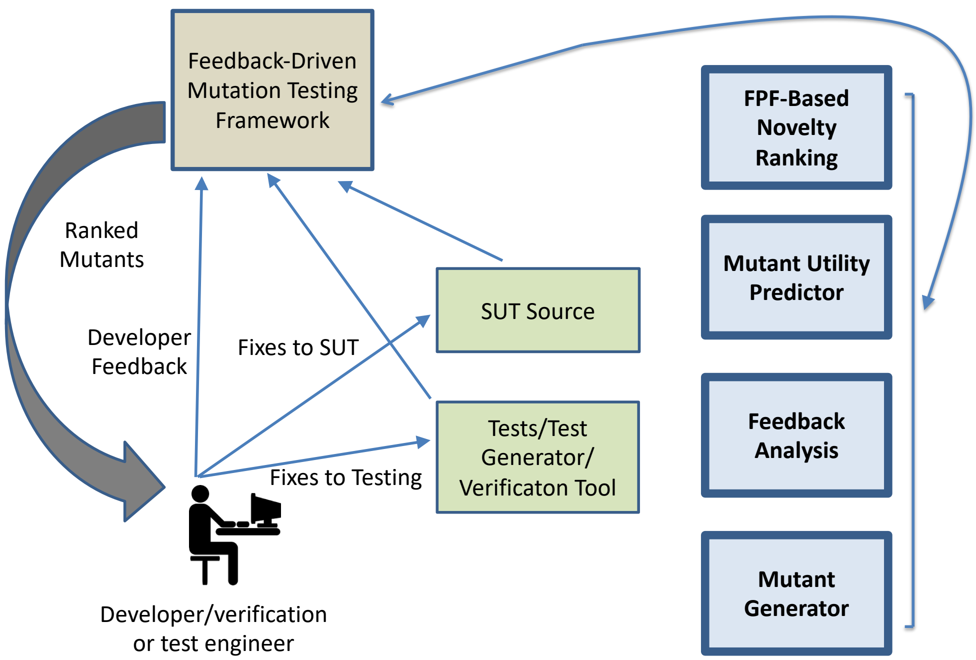

Figure 2: Basic flow of feedback-driven mutation testing.

comparable to PIT [20] and Major [68], despite not parsing the Java program at all, or acting at the bytecode level. Furthermore, we showed that for falsification-driven verification, it produced mutants of equal value to those produced by Andrews' tool [6] and Muupi [81] for C and Python, respectively. As it stands, the `universalmutator` tool is useable for real-world mutation (in fact, it is being considered for use by NASA/JPL engineers in testing the C code for upcoming CubeSat missions. It allows easy definition of new rules, and supports automated analysis of mutants, coverage-based pruning of mutants, and (in some languages) trivial compiler equivalence [100] checks. As it stands, the tool and approach form a suitable platform for generating mutants to be used in this project's early phases. Below, we discuss enhancements to the approach and tool that will be required to better support robust, flexible, efficient feedback-driven mutation testing.

3 Research Plan

3.1 Overview

Figure 2 shows the basic outline of a proposed workflow and components needed to support feedback-driven mutation testing. These components serve to organize our research plan.

3.1.1 FPF-Based Novelty Ranking

Ranking unkilled mutants according to how much "new" information they might provide to users requires more than simply using the FPF algorithm. FPF requires a distance metric, and a distance metric requires a *representation* of mutants. Mutants can be similar because they modify the same line, function, class, or module, but also because, despite being located in very different parts of a program, they are very semantically similar. E.g., a mutant to the parser of a compiler, to an I/O error-handling routine in the code generator, and to a complex optimization pass may all be very "similar" in the only meaningful sense if all three mutants modify logging statements that don't have any actual effect on the state of the compiler. Figure **??** shows the fundamental problem. Elements of the distance metric must include mutant location, mutation operator, and some representation of the code element modified — language construct, functions called, variables modified, and so forth. This is a complex problem in representation and weighting of elements of a representation, especially if we aim at producing a language and project agnostic metric, perhaps one open to tuning via feedback analysis. We plan to exploit metric learning methods [72], but are aware that to avoid over-fitting to even a set of good examples, the final metric may have to be hand-tuned. In part this is due to the difficulty of establishing large amounts of ground truth data; although there are unsupervised approaches to metric learning [107, 115], the most popular approaches require supervision.

3.1.2 Mutant Utility Predictor

3.1.3 Feedback Analysis

3.1.4 Mutant Generator

3.2 Core Research Questions

3.2.1 Research Questions for Feedback-Driven Mutation Testing in General

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 balance FPF-based selection of mutants for novelty with predictions of mutant equivalence, outcome, dominance, and productivity?
4. How can we identify outliers in otherwise similar groups of mutants, and is such identification useful?
5. How can we set budgets for automated test generation and timeouts for verification efforts to quickly estimate whether a mutant is killable?
6. Can we predict whether a mutant's unkillability is due to poor test generation or due to oracle weakness?
7. How can we most effectively use already generated killing tests and counterexamples to prune mutants?
8. 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)?

3.2.2 Research Questions Specific 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)?

3.3 Work and Evaluation Plans

3.3.1 Work Plan

For this project, a simple plan should suffice, defined by components of Figure 2. In Year One, PI Groce and the graduate student will both focus on the FPF-Based Novelty Ranking and representation of mutants and test results. In Year Two, both will focus on the Mutant Utility Predictor and Feedback Analysis, and how to integrate those with the FPF-Based Novelty Ranking. In Year Three, the focus will shift to the Mutant Generator itself, and the problem of any-language mutant generation. The third year will also emphasize integrating the entire system and performing more thorough evaluations.

3.3.2 Evaluation Plan

4 Related Work

4.1 Bug Triage and Distance Metrics in Software Engineering

A fundamental insight of this proposal is that the problem of presenting mutants to a user is conceptually similar to the "fuzzer taming"/crash bucketing/bug triage problem studied in automated testing research [19, 117]. Our approach is centered on the idea of computing distances between mutants. The use of distance metrics in software engineering for a variety of purposes is long-standing. To our knowledge, almost all such uses are essentially spectrum-based [106] (that is, using counts of coverage of code entities), except for some work in model-checking [37, 15] and some of the metrics in our recent fuzzer taming work [19]. Reneiris and Reiss initially proposed using distance between executions to localize faults [105], and Liu and Han [80], among others have followed this line with various metrics (in their case, using the localization of a spectrum-based method to determine distance). Vangala et al. proposed using distance to cluster test cases to improve diversity and eliminate duplicates [118]. Methods for clustering to identify bugs all rely on an implicit distance determined by a feature set [28]. Our own previous work has used distance metrics for a variety of purposes [19, 46, 126]. Adaptive random testing uses distances (usually between inputs, not executions, however) to choose tests [17, 18, 9]. Recently, we proposed a novel, causal metric for fuzzer taming and fault localization, itself based on mutation testing results [58]. More generally, work on presenting a few mutants to maximize human efforts to improve a testing or verification effort is related to our own work on end-user testing of machine learning systems [71, 48], where the limits of human patience are considered as a key factor in a system for presenting actionable information to a user.

4.2 Mutation Testing

There is a vast body of work on mutation testing or analysis. Mathur attributes [83] 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. [22], and the approach was first implemented by Budd et al. [14] 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 [119, 120, 34, 94, 95, 75, 34, 31]. In this project, we are more interested in the practical effectiveness of mutation testing than in theoretical justifications.

It has long been argued [13] that mutation analysis is *stronger* than other coverage measures. The subsumption of multiple coverage measures by mutation analysis, including all the basic coverage measures [91] was shown by Offutt [97], and data flow subsumption was demonstrated by Mathur [85]. Daran et al. [21] found that mutation analysis produces faults that are similar to actual faults in terms of the error traces produced. Andrews et al. [7, 8] 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. [69] 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 [101] 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. Ahmed et al., using a very different approach, reached a similar conclusion about the potentially high value for suite improvement, but weak correlation of mutation analysis results [2].

Cost of execution is often [67] 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 [96] 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 [13, 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 [82], Wong et al. [121, 122], and Offutt et al. [98].

Determining relative merits of selective mutation strategies such as operator selection and random sampling has long been an active field of research [122, 90, 130] Skew in fault representativeness among mutants was initially noticed by Budd et al. [13] who found that particular types of mutants are representative for particular kinds of faults. Constrained mutation was pioneered by Mathur [82, 84] and was further investigated by Wong et al. [123]. An extension of this approach called *n*-selection was suggested by Offutt et al. [98] 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.[10]. Namin et al. [92, 93] formulated the concept of *sufficient mutation operators*, and Untch [116] even proposed simply using statement deletion as "the" mutation operator. Deng et al. [23] 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 [32, 109, 79]. Higher order mutants (HOM) aim to improve the quality of mutants by combining simpler mutants into more complex mutants. Jia et al. [66, 65], 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[24, 112, 62] 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 [130, 129]. Zhang et al. [127] 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 [33] 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 [36]. Recent work on Predictive Mutation Testing (PMT) [128] applies machine learning to build a model that can predict mutation results without actually running mutation testing, a novel and promising *do-smarter* approach.

4.2.1 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 [49, 53, 3].

Very recent work by Papadakis et al. (some unpublished in a conference of journal at the time this proposal was written) has aimed, unusually, at predicting the "quality" of [99] or even prioritizing [16] mutants, to rank fault-revealing mutants highly so that users can produce tests to find faults. This work is highly relevant to our aims, but focuses on a single static pass to rank mutants by their fault-revealing potential, informed by (possibly cross-project) data on fault-revealing tests. There is no feedback loop, or ability to indicate the importance of various faults in the program under test, and the language support issue is dodged by targeting LLVM bitcode. Nonetheless, the goals, methods, and results of this work will inform our own approach, in particular in the utility estimation aspect that balances FPF-determined novelty.

The single most relevant work by others, however, is to be found in a report on actual techniques in use at Google for applying mutation testing to real-world projects [102]. 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 [63], 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.

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 [54] and Holzmann's work on lightweight textual code analysis with Cobra [60]. Trivial compiler equivalence [100] is a key technology for supporting effective any-language mutation; especially in the context of this proposal, where executing all mutants is not the goal, simply letting the compiler handle many validity and equivalence questions is a simple and effective approach.

5 Broader Impacts

Improving Software System Reliability: A key element of broader outreach will be to report bugs discovered during our testing experiments, and contribute improved test suites to critical open source projects. To that end, we will primarily target real world systems in our experiments, in hopes of improving their quality, and the quality of their testing. While we expect to develop more examples, our current infrastructure includes automated testing for the Linux kernel RCU module, Google and Mozilla JavaScript engines, a variety of C compilers (including GCC and LLVM), YAFFS2 [124] and other file systems, Google's Go compiler, a large set of Unix utilities, and a large number of Python libraries (including some of the most widely used libraries, and key scientific and numeric analysis packages). In previous work, discussions with working test engineers at Mozilla, Google, and NASA have significantly informed our research progress, and we expect this to continue. In the long term, we believe that a mutation-driven development paradigm might result in easier development of critical software components in tandem with an extremely high-quality, specification-defining automated test suite.

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 [56] 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:** The results of CCF-1217824 included a preliminary exploration of how to "tame" fuzzer output, a problem also central to this proposal [19]. In previous work, the goal was to find an algorithm for using hand-chosen distance metrics to identify bugs in tests; in this proposal, other methods for taming fuzzers are addressed. A key result from CCF-1217824 is the development of a strategy for creating "quick tests" [46], 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 [52, 126]. Moreover, benefits do not depend on 100% preservation of a property [5]. Other results include an overview of the value of coverage in testing experiments [45] and exploration of how individual test features impact the coverage and fault detection statistics of random tests [44]. 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 [77, 74, 26, 78]. **Research Products:** Several publications resulted from this grant, including those cited above and numerous others [45, 19, 126, 46, 44, 4, 52, 59, 50, 5, 59, 39], along with three PhD theses. Source code [40, 51] is available on GitHub.

References

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

[2] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. Can testedness be effectively measured? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 547–558, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi:10.1145/2950290.2950324. URL `http://doi.acm.org/10.1145/2950290.2950324`.

[3] 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.

[4] 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`.

[5] 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.

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

[7] 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.

[8] 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.

[9] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness. In *International Symposium on Software Testing and Analysis*, pages 265–275, 2011.

[10] 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.

[11] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

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

[14] 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.

[15] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering*, pages 73–82, 2004.

[16] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *CoRR*, abs/1803.07901, 2018. URL `http://arxiv.org/abs/1803.07901`.

[17] Tsong Yueh Chen. Fundamentals of test case selection: diversity, diversity, diversity. In *International Conference on Software Engineering and Data Mining*, pages 723–724, 2010.

[18] Tsong Yueh Chen, Hing Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer*

Science, pages 320–329, 2004.

[19] 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.

[20] Henry Coles. Pit mutation testing. `http://pittest.org/`.

[21] 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.

[22] 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.

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

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

[25] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012. ISSN 1045-9219. doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159.

[26] 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.

[27] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[28] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying software failures. In *International Symposium on Software Reliability Engineering*, pages 451–462, 2004.

[29] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[30] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Does choice of mutation tool matter? *Software Quality Journal*, 25(3):871–920, September 2017. ISSN 0963-9314. doi:10.1007/s11219-016-9317-7. URL `https://doi.org/10.1007/s11219-016-9317-7`.

[36] 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.

[37] Alex Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.

[38] Alex Groce and Martin Erwig. Finding common ground: Choose, assert, and assume. In *International Workshop on Dynamic Analysis*, pages 12–17, 2012.

[39] Alex Groce and Jervis Pinto. A little language for testing. In *NASA Formal Methods Symposium*,

pages 204–218, 2015.
[40] Alex Groce, Yang Chen, John Regehr, Chaoqiang Zhang, and Amin Alipour. Swarmed versions of random testing tools. https://github.com/agroce/swarmed_tools, .
[41] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. Regexp based tool for mutating generic source code across numerous languages. `https://github.com/agroce/universalmutator`, .
[42] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.
[43] Alex Groce, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Workshop on Constraints in Formal Verification*, pages 1–15, 2008.
[44] 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.
[45] 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`.
[46] 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.
[47] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.
[48] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret Burnett, Weng-Keen Wong, Simone Stumpf, Shubhomoy Das, Amber Shinsel, Forrest Bice, and Kevin McIntosh. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. *IEEE Transactions on Software Engineering*, 40(3):307–323, March 2014. ISSN 0098-5589. doi:10.1109/TSE.2013.59.
[49] 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.
[50] 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.
[51] 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.
[52] 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*, 26(1):40–68, 2016.
[53] 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*, 25(4):917–960, 2018.
[54] 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.

[55] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, May 2008.

[56] 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.

[57] Farah Hariri and August Shi. SRCIROR: a toolset for mutation testing of C source code and LLVM intermediate representation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 860–863, 2018. doi:10.1145/3238147.3240482. URL http://doi.acm.org/10.1145/3238147.3240482.

[58] Josie Holmes and Alex Groce. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *IEEE International Symposium on Software Reliability Engineering*, 2018.

[59] 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*, 20(1):57–78, 2018.

[60] 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.

[61] Ivy Hooks. Writing good requirements. In *INCOSE International Symposium*, volume 4, pages 1247–1253. Wiley Online Library, 1994.

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

[63] 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.

[64] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[65] 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.

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

[67] 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.

[68] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE*, 2011.

[69] 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.

[70] Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[71] Todd Kulesza, Margaret M. Burnett, Simone Stumpf, Weng-Keen Wong, Shubhomoy Das, Alex Groce, Amber Shinsel, Forrest Bice, and Kevin McIntosh. Where are my intelligent assistant's mistakes? A systematic testing approach. In *End-User Development - Third International Symposium, IS-EUD 2011, Torre Canne (BR), Italy, June 7-10, 2011. Proceedings*, pages 171–186, 2011. doi:10.1007/978-

3-642-21530-8_14. URL https://doi.org/10.1007/978-3-642-21530-8_14.

[72] Brian Kulis. Metric learning: A survey. *Foundations & Trends in Machine Learning*, 5(4):287–364, 2012.

[73] Imre Lakatos. The role of crucial experiments in science. *Studies in History and Philosophy of Science Part A*, 4(4):309–325, 1974.

[74] 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.

[75] 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.

[76] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 429–432. ACM, 2014.

[77] 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.

[78] 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.

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

[80] Chao Liu and Jiawei Han. Failure proximity: a fault localization-based approach. In *Foundations of Software Engineering*, pages 46–56, 2006.

[81] Xin Liu. muupi mutation tool. https://github.com/apepkuss/muupi.

[82] 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.

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

[84] 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.

[85] 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.

[86] Paul E. McKenney. RCU torture test operation. https://www.kernel.org/doc/Documentation/RCU/torture.txt.

[87] Paul E. McKenney. Structured deferral: synchronization via procrastination. *Commun. ACM*, 56(7): 40–49, July 2013. ISSN 0001-0782. doi:10.1145/2483852.2483867. URL http://doi.acm.org/10.1145/2483852.2483867.

[88] Mark Mossberg. https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/, April 2017.

[89] mrbean bremen, jmcgeheeiv, et al. pyfakefs implements a fake file system that mocks the Python file system modules. https://github.com/jmcgeheeiv/pyfakefs, June 2011.

[90] 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.

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

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

[93] 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.

[94] 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.

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

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

[97] 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.

[98] 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.

[99] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 32–39, April 2018. doi:10.1109/ICSTW.2018.00025.

[100] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*, pages 936–946, 2015.

[101] 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.

[102] 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`.

[103] Karl Popper. *The Logic of Scientific Discovery*. Hutchinson, 1959.

[104] Karl Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. 1963.

[105] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, 2003.

[106] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference Held Jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, 1997.

[107] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[108] Kostya Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity, Development (SecDev), IEEE*, pages 157–157. IEEE, 2016.

[109] 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.

[110] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[111] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan

Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, 2016.

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

[113] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[114] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests with Unit Meister. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 241–244, 2005.

[115] Michael E Tipping and Christopher M Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.

[116] 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.

[117] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 612–622, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. doi:10.1145/3238147.3238200. URL `http://doi.acm.org/10.1145/3238147.3238200`.

[118] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. In *ESEC/FSE*, pages 293–294, 2009.

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

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

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

[122] 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.

[123] 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.

[124] Yaffs. Yaffs: A flash file system for embedded use. `http://www.yaffs.net/`.

[125] Michal Zalewski. american fuzzy lop (2.35b). `http://lcamtuf.coredump.cx/afl/`, November 2014.

[126] 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.

[127] 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.

[128] 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.

[129] 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.

[130] 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.