Guiding Testing Effort Using Mutant Utility

Justin Alvin
University of Massachusetts
Amherst, MA, USA
jalvin@umass.edu

Bob Kurtz, Paul Ammann, Huzefa Rangwala
George Mason University
Fairfax, VA, USA
{rkurtz2, pammann}@gmu.edu, rangwala@cs.gmu.edu

René Just
University of Washington
Seattle, WA, USA
rjust@cs.washington.edu

***Abstract*—This paper addresses two long-standing goals in software testing: making mutation-based testing practical and software testing overall more effective, predictable, and consistent. To that end, this paper proposes a novel mutation-based approach that guides testing effort based on test goal utility.**

I. Introduction

Writing effective software tests is a challenging task, in particular in the absence of well-defined test goals. A test is *effective* if it detects faults, but the set of faults in a program is unknowable. Hence, developers cannot use this definition and instead rely on proxies for test effectiveness.

An established proxy for test effectiveness in practice is the code coverage ratio, where each *test goal* is a directive to execute, e.g., a particular statement or branch in the code. Code coverage is intuitive, cheap to compute, and well supported by commercial tools. However, coverage-adequate test suites, which satisfy all test goals, are not the norm and neither should they be [1]. In practice, developers only satisfy a fraction of the test goals, but adequate thresholds for code coverage ratios are inherently arbitrary and a matter of much debate [2].

An established proxy for test effectiveness in research is the mutant detection ratio [3], [4], which measures a test's ability to distinguish a program under test from many artificial faults, called mutants. Mutation-based testing, in which mutants are test goals, has a key advantage compared to coverage-based testing: it elicits test assertions, which ensure that a test not only executes the code but also asserts on its behavior.

Mutation-based testing is a rigorous approach to software testing, but comes at a substantial cost in terms of number of test goals. Further, the mutant detection ratio, as a proxy for test effectiveness, shares exactly the same limitations as the code coverage ratio. While mutation-based testing generates test goals that elicit effective tests, it inevitably generates additional test goals that do not [5]. The latter should not be satisfied, but the mutant detection ratio does not provide any guidance to distinguish them. Likewise, the mutant detection ratio is not suitable for comparing the testedness of different artifacts because the ratio of test goals that should be satisfied differs across them. Which test goals to satisfy (first) and when to stop testing, heavily relies on developer experience and characteristics of the program under test—neither of which is captured by the mutant detection ratio.

We argue that a useful proxy for test effectiveness must include a notion of *test goal utility* (i.e., the likelihood that a test goal elicits an effective test) and must be *context sensitive* (i.e., consider test goal utility in the context of a given artifact).

Test goal utility enables *effective* and *predictable* testing: developers can make informed decisions about which test goals to satisfy and when to stop. Context sensitivity enables *consistent* testing: developers can reason about test goals in context and assess whether test goals in similar context are consistently satisfied (or not) across the codebase.

This paper describes a mutation-based approach that aims at making mutation-based testing practical and software testing overall more effective, predictable, and consistent. There are two key insights behind this approach. First, the question of what constitutes an adequate threshold for mutant detection ratios is ill-posed because it presupposes that each test goal is equally important to the developer. This assumption is invalid, and what developers need is specific guidance as to which test goals to satisfy—not just how many. Second, an existing codebase, both programs and tests, encode developer expertise that can usefully differentiate test goals that should be satisfied from test goals that should not, based on program context.

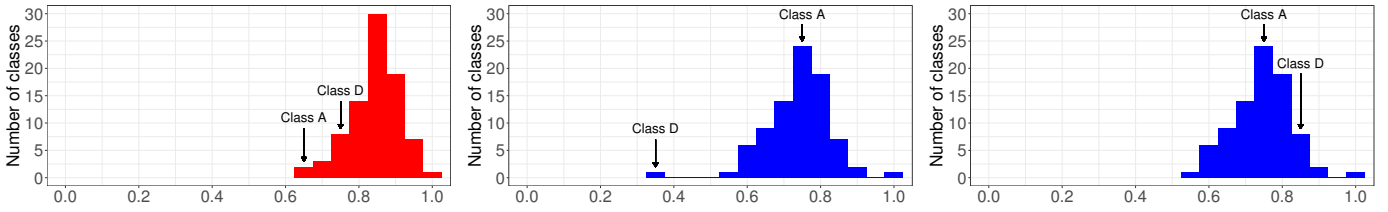
II. Predicting Mutant Utility from Program Context

Our approach extends the work of Just et al. [6], who showed that mutant utility correlates with program context. Specifically, our approach exploits these correlations and extracts syntactic and semantic program context features from a program's abstract syntax tree to train a classifier that can predict the utility of individual mutants [7].

III. Effective, Predictable, and Consistent Testing

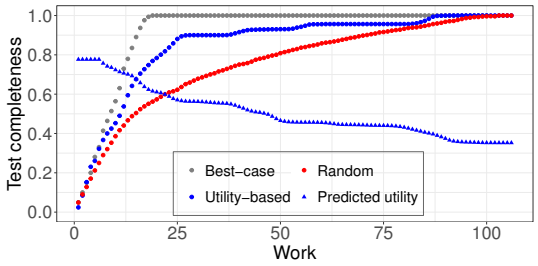
We evaluated our approach using 85 unique classes from the Defects4J corpus [8], each accompanied by a thorough developer-written test suite. We used this set of well-tested classes to predict mutant utility in a cross-validation setting, and to simulate mutation-based testing. Further analyses showed that even a training set with as few as 10 classes yields reasonable accuracy for within-project predictions and that cross-project prediction accuracy is acceptable for training and incrementally refining a generic classifier [7].

Effective testing Figure 2a shows the outcome from a mutation-based testing simulation for a single class, for which the trained classifier achieves average performance. Each unit of work represents a mutant presented to a developer and test completeness is measured as the ratio of detected dominator mutants, which form a set of non-redundant mutants that subsume all other mutants. Mutant-detection ratios are measurable but inherently inflated due to redundancy among mutants, and hence a poor proxy of test completeness. In contrast, the

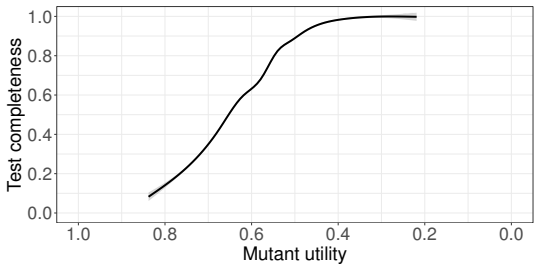


(a) Mutant-detection ratio, existing code base. (b) Our approach (AUC), existing code base. (c) Our approach (AUC), after adding two missing tests for Class D to the code base.

Fig. 1. Consistent testing: prediction performance for mutant utility can identify test deficiencies, whereas mutant-detection ratios can be misleading.



(a) Effective testing: test completeness vs. work.



(b) Predictable testing: mutant utility as a stopping criterion.

Fig. 2. Mutant utility enables effective testing with predictable test completeness.

ratio of detected dominator mutants is a great proxy for test completeness, but cannot be computed a priori [7].

Best-case shows the effect of ground-truth selection of dominator mutants at each step, and hence not wasting any work. *Random* is averaged over 100 iterations and shows the expected test completeness against work for random mutant selection, which is the state of the art and simulates what a developer would currently face in practice. Finally, *Utility-based* shows our approach—ranking mutants by *predicted utility*.

Figure 2a shows: (1) The utility-based curve tends to rise early at nearly the same pace as the best-case curve, and much more steeply than the random curve. (2) Our approach correctly identifies high-utility mutants (predicted mutant utility above 0.6), which elicit effective tests. It also correctly identifies low-utility mutants (predicted utility below 0.4), which a developer should ignore. (3) The desirable trend for the utility-based curve only holds until a test completeness of about 90% and then there are three visible plateaus, which are mutants misclassified due to uncertainty. We expect that active learning and context-model refinements will resolve the uncertainty and further close the gap to best-case selection.

Predictable testing Figure 2b shows the association between mutant utility thresholds and test completeness (averaged over 85 classes). Figure 2b suggests that the predicted mutant utility provides, for the first time, a stopping criterion for mutation-based testing that is computable a priori and strongly correlated with test completeness. The expected test completeness is about 65% for a utility threshold of 0.6 and approaches 100% for 0.4. Based on requirements and resources, a developer can make an informed decision about when to stop testing.

Consistent testing Figure 1a shows that the mutant-detection ratio can be misleading when reasoning about test deficiencies. Manually determined ground truth revealed that class A's test suite is adequate, whereas class D's test suite is deficient. Specifically, most of the undetected mutants in class A cannot or should not be detected. Considering undetected mutants with the exact same context in all other 84 classes, these mutants are almost never detected, which suggests that the developer correctly avoided writing tests for them in class A. For class D, the opposite is true—many of its undetected mutants are almost always detected in all other 84 classes.

Our approach reasons about the prediction performance of the trained classifier, in a hold-one-class-out setting [7]. More precisely, it measures AUC (area under the ROC curve: < 0.5 indicates inverse predictions). If the classifier frequently makes wrong predictions (i.e., has an unusually low AUC value) about whether mutants should be detected in a particular class, then this indicates an inconsistency: mutants in similar context in other classes are not similarly tested.

In contrast to the mutant-detection ratio, our approach correctly identifies the test-deficient class D (Figure 1b). We also added two missing tests for class D (Figure 1c): the anomaly disappears and all classes appear fairly consistently tested.

IV. Conclusions

This paper proposes a novel, mutation-based approach that guides testing effort based on test goal utility. It motivates the approach and reports on promising results. Future work includes incorporation of active learning strategies and refinement of the proposed program context model.

References

[1] B. Marick, "How to misuse code coverage," in *Proc. of ICTCS*, 1999.
[2] A. Savoia, "Code coverage goal: 80% and no less!" July 2010, https://testing.googleblog.com/2010/07/code-coverage-goal-80-and-no-less.html.
[3] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. of FSE*, 2014, pp. 654–665.
[4] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of ICSE*, 2005, pp. 402–411.
[5] G. Petrović, M. Ivanković, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *Proc. of Mutation*, 2018, pp. 47–53.
[6] R. Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proc. of ISSTA*, 2017, pp. 284–294.
[7] B. Kurtz, "Improving mutation testing with dominator mutants," Ph.D. dissertation, George Mason University, Fairfax, VA, December 2018.
[8] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. of ISSTA*, 2014, pp. 437–440.