Using Fault History to Improve Mutation Reduction

Laura Inozemtseva
University of Waterloo
Waterloo, ON, Canada
lminozem@uwaterloo.ca

Hadi Hemmati
University of Manitoba
Winnipeg, MB, Canada
hemmati@cs.umanitoba.ca

Reid Holmes
University of Waterloo
Waterloo, ON, Canada
rtholmes@uwaterloo.ca

ABSTRACT

Mutation testing can be used to measure test suite quality in two ways: by treating the kill score as a quality metric, or by treating each surviving, non-equivalent mutant as an indicator of an inadequacy in the test suite. The first technique relies on the assumption that the mutation score is highly correlated with the suite's real fault detection rate, which is not well supported by the literature. The second technique relies only on the weaker assumption that the "interesting" mutants (i.e., the ones that indicate an inadequacy in the suite) are in the set of surviving mutants. Using the second technique also makes improving the suite straightforward.

Unfortunately, mutation testing has a performance problem. At least part of the test suite must be run on every mutant, meaning mutation testing can be too slow for practical use. Previous work has addressed this by reducing the number of mutants to evaluate in various ways, including selecting a random subset of them. However, reducing the set of mutants by random reduction is suboptimal for developers using the second technique described above, since random reduction will eliminate many of the interesting mutants.

We propose a new reduction method that supports the use of the second technique by reducing the set of mutants to those generated by altering files that have contained many faults in the past. We performed a pilot study that suggests that this reduction method preferentially chooses mutants that will survive mutation testing; that is, it preserves a greater number of interesting mutants than random reduction does.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Measurement, Performance

Keywords

Mutation testing, mutant reduction, fault history, test suite quality

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2237-9/13/08
http://dx.doi.org/10.1145/2491411.2494586

1. INTRODUCTION

Mutation testing offers software developers a way to evaluate the quality of their test suite. In mutation testing, *mutants* are created by making small syntactic changes to a program, which we will refer to as the *system under test (SUT)*. For example, a mutant could be created by modifying a constant, negating a branch condition, or removing a method call. The resulting mutant may produce the same output as the original SUT, in which case it is called an *equivalent mutant*. As or after the mutants are generated, the program's test suite is run on each one. If the suite fails, it is said to have *killed* the mutant. The number of mutants the suite kills, divided by the total number of non-equivalent mutants, is the suite's *kill score*. Equivalent mutants are excluded because they cannot, by definition, be detected by an automated test.

There are two ways a developer can use the output of mutation testing:

TECHNIQUE 1. *The developer treats the kill score as a quality metric, where higher is better. This technique assumes that the mutant kill score is correlated with the suite's actual fault detection ability.*

TECHNIQUE 2. *The developer ignores the kill score and focuses on the set of surviving mutants. Each non-equivalent surviving mutant indicates an inadequacy in the test suite that the developer may want to address.*

The downside of Technique 1 is the built-in assumption that the mutation score is highly correlated with the suite's real fault detection rate. This assumption has only been tested in two studies [1, 4], both of which used a single small subject program. It is therefore not clear that it holds in general. This technique also does not help the developer write new tests: there is no obvious way to use the mutation score to guide improvements to the suite.

Technique 2, by contrast, is less reliant on the assumption that the mutation score is highly correlated with the suite's real fault detection rate. Treating a surviving, non-equivalent mutant as an indicator of an inadequacy in the test suite does assume that the mutant is at least somewhat representative of a real fault. However, it does not require making any assumptions about the killed mutants; we merely assume that the interesting mutants will be found in the set of surviving mutants. Trivial mutants, which are one of the worries of mutation testing, will be killed easily and the developer will not need to consider them. Moreover, the extra context the developer gets from manually investigating the mutants makes this technique much less reliant on there being a strong correlation between the kill

score and the fault detection ability of the suite. This technique also makes improving the suite straightforward: the developer simply writes tests that can kill the interesting mutants. These advantages make Technique 2 the preferable approach.

Unfortunately, mutation testing has a drawback that applies to both techniques. Since it requires generating tens of thousands of mutants for a typical SUT, and at least part of the test suite must be executed for each mutant to determine if it can be killed, mutation testing can be too slow for practical use. It is possible to speed up mutation testing with techniques like changing the definition of "killing" a mutant, evaluating mutants[1] in parallel, or computing the results incrementally as the program evolves. However, as long as the same number of mutants are being evaluated, there is a limit to how much speedup can be achieved. In particular, even though mutation testing is embarrassingly parallel, parallelization is insufficient on its own because of the large number of mutants.

As we will describe in Section 2, previous studies have explored improving the performance of mutation testing through mutant reduction methods. These studies either used fewer mutation operators, and thus generated fewer mutants, or selected a random subset of the generated mutants to evaluate. While these approaches may be acceptable for a developer using Technique 1, they are suboptimal for a developer using Technique 2, because they will eliminate many mutants that would survive if they were evaluated. Each of these mutants indicates an inadequacy in the test suite, so eliminating them prevents the developer from identifying those inadequacies. It would be preferable to reduce the mutant set in a way that preserves as many interesting mutants as possible.

We propose a novel reduction method that supports the use of Technique 2. More precisely, we propose evaluating only the *faulty-file mutants*, or mutants that are generated by altering a file that has contained many faults in the past. This reduction method has three desirable characteristics:

1. **Improves Performance:** Since previous work has shown that a small number of classes in an object-oriented program contain most of the faults [6], and Java programs typically have one class per file, this method will greatly reduce the number of mutants that need to be evaluated, improving performance.[2]
2. **Focuses Developer Effort:** Since the number of faults that have been found in a file in the past is a good predictor of the number of faults that will be found in the file in the future [2], focusing the developer's effort on the faulty files and the interesting mutants that are generated from them is likely to be a good use of the developer's time.
3. **Supports the Use of Technique 2:** Since faulty files by definition are associated with many of the faults that were found in the program, we hypothesized that faulty-file mutants would be more likely to survive mutation testing. This means that our reduction method would eliminate fewer interesting mutants than random reduction, supporting the use of Technique 2.

[1]We use the phrase "evaluate a mutant" to refer to running the program's test suite on the mutant version of the program to determine if the suite can kill the mutant.

[2]Note that the reduction is done before the mutants are evaluated: there would be no performance benefit otherwise.

We explored the validity of the third reason in a pilot study that we will describe in Section 3. Our results suggest that our reduction method supports the use of Technique 2 by preferentially choosing mutants that will survive mutation testing. Section 4 describes the full study we plan to perform to confirm and extend these results. Section 5 indicates our desired feedback and Section 6 concludes the paper.

2. RELATED WORK

Previous attempts to speed up mutation testing have used the following four approaches:

1. Weakening the definition of "killing" a mutant [7, 15];
2. Accelerating the testing process by, for instance, evaluating mutants in parallel or adding compiler support for mutation testing, e.g., [5, 11, 13];
3. Computing the mutation testing results incrementally [17]; and
4. Selecting a subset of the mutants to evaluate [3, 9, 10, 14, 16].

Our approach is complementary to the first three. For example, a developer could reduce the set of mutants using our method and then evaluate the remaining mutants in parallel. The fourth approach is the one we use, so we describe these studies in more detail.

In 1995, Wong and Mathur [14] proposed reducing the set of mutants by randomly selecting a subset of them to evaluate. They found that the number of mutants could be reduced significantly with little impact on the results. Later work focused on selective mutation, or reducing the number of mutants by reducing the number of mutation operators used to generate the mutants. Specifically, in 1996, Offutt et al. [10] reduced the set of 22 operators used by Mothra, a mutation tool for Fortran programs, to 5 operators, and showed that there was little change in the results. In 2001, Barbosa et al. [3] developed guidelines for the determination of a sufficient set of mutation operators. In 2008, Namin et al. [9] used variable reduction to identify 28 key mutation operators for the C language. Finally, in 2010, Zhang et al. [16] compared random reduction and selective mutation. They found that operator-based mutation selection is not superior to random selection.

2.1 Novelty of the Idea

The single most related paper by the same authors is [8], which uses mutation testing to explore the relationship between a test suite's size, its coverage, and its fault detection effectiveness. It does not study the practical aspects of mutation testing, including efficiency, at all.

The single most related paper by other researchers is Wong and Mathur's work [14], since they also reduce the set of mutants by selecting a subset to evaluate. Unlike them, we do not select the subset randomly. While random selection is simple, it is suboptimal for developers using Technique 2.

The other studies we described used operator-based reduction. While useful, this technique has limits: modern mutation tools such as PIT[3] use a small number of operators to begin with (10 in PIT's case), so it is unlikely that the operator sets can be reduced much further. Moreover, eliminating whole classes of mutants may eliminate the mutants that would have survived mutation testing.

[3]`http://pitest.org/`

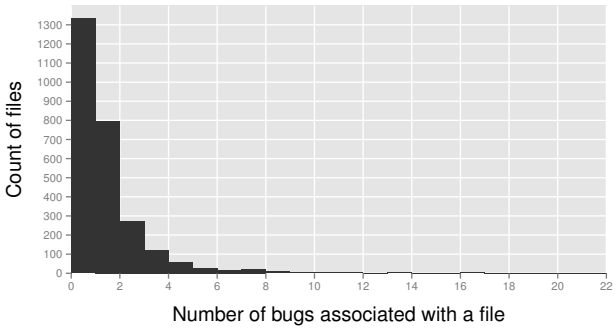

Figure 1: A histogram of the number of bug IDs associated with Java files from Apache POI. The x axis gives the number of bug IDs, while the y axis gives the number of files that are associated with a specific number of bug IDs.

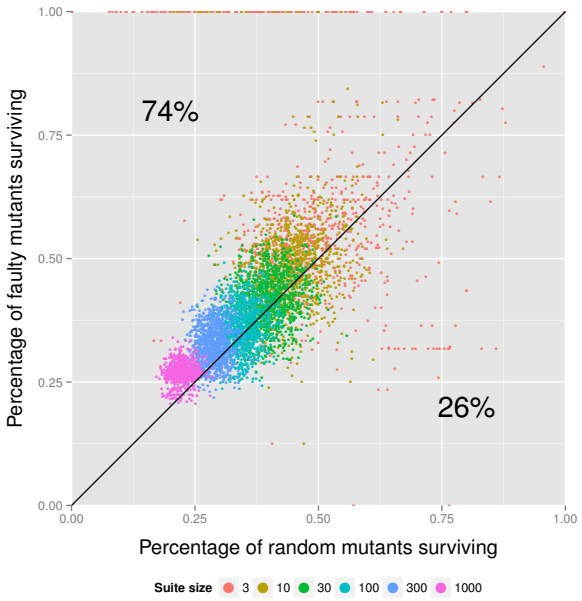

Figure 2: The relationship between the fraction of randomly selected mutants that survive, p_r, and the fraction of faulty mutants that survive, p_f, for 6,000 test suites of various size.

To the best of our knowledge, this study is the first attempt to reduce the number of mutants in a non-random way without changing the mutation operators, making it a novel approach to improving the performance of mutation testing. We summarize the new idea as follows:

Not all mutants are equal. Evaluating only faulty-file mutants will improve the performance of mutation testing, as random mutant selection does, but will preserve a greater proportion of interesting mutants.

3. PILOT STUDY

The goal of the pilot study was to determine if the faulty-file reduction method preferentially selects mutants that will survive mutation testing.

We began by choosing a subject program: Apache POI[4], an open source API for Microsoft Office documents that contains approximately 295,000 source lines of Java code.

[4]`https://poi.apache.org/`

Table 1: The percentage of suites that preferentially choose mutants that would survive (i.e., the percentage above the line $p_f = p_r$), grouped by suite size.

Size	% of Points Above Line
3	76.9
10	68.6
30	60.2
100	59.3
300	79.7
1,000	97.2
All sizes	73.7

Using standard techniques [12] we combined information from POI's version control repository and its bug tracker to identify historically faulty files. As Figure 1 shows, the majority of the Java files in the program (2,131 of 2,691) are associated with zero or one bug IDs. Six files are associated with more than 20 bug IDs; these are not shown in the figure for space reasons. Two of the six files are part of the regression test suite, so it is not surprising that they are associated with a large number of past faults. We ignored these two files, since test files are not mutated, and considered only the remaining four faulty files.[5]

Once we had identified the faulty files, we used the mutation tool PIT to perform standard mutation testing on Apache POI. PIT generated 27,720 mutants for POI; of these, 776 or 3% were made by mutating one of the four faulty files that we identified. This verifies that our reduction technique greatly decreases the number of mutants that need to be evaluated.

Next, we compared our reduction method to random reduction as follows. For $s \in \{3, 10, 30, 100, 300, 1000\}$, we made 1,000 test suites of size s by randomly selecting test methods without replacement from Apache POI's full suite. In other words, we created a total of 6,000 random test suites of varying size. For each suite t, we ran the suite on the set of faulty-file mutants to determine the percentage of mutants that survived, p_f. Note that we include equivalent mutants in the set of surviving mutants. We then selected 776 mutants randomly from the 27,720 that PIT generated and ran t on the random mutants to determine the percentage of mutants that survived, p_r. Figure 2 shows the results. Each point in the figure represents p_f and p_r for one test suite. If our reduction method did not preferentially choose mutants that would survive, we would expect the points to be equally distributed above and below the line $p_f = p_r$. Instead, we see that the points tend to lie above this line. Specifically, 4,419 of the 6,000 test suites, or 74%, have $p_f > p_r$. This means that for 74% of the suites, more mutants survive when we use the faulty-file reduction method than when we use the random reduction method. We break down this result by suite size in Table 1. As the table shows, the percentage of suites above the $p_f = p_r$ line initially drops as suite size increases, but rises to 97% for the 1,000 method suites.

[5]The threshold of 20 bug IDs was set arbitrarily based on manual inspection of the histogram; we will evaluate alternative, automated methods of choosing an appropriate threshold in our full study.

Finding: our current results suggest that our reduction method preferentially chooses mutants that will survive mutation testing.

4. PLANNED FULL STUDY

Our full study will do the following:

- Confirm the findings of our pilot study by extending it to more test subjects;
- Confirm the performance improvement by measuring runtime for both standard and faulty file mutation testing;
- Explore alternate methods of predicting which mutants will survive, since fault data may not always be available; and
- Explore other ways of choosing a threshold for faulty files.

In addition to extending the pilot study, we are interested in exploring how this reduction method can be used in higher order mutation testing, where the performance issue becomes even more pressing.

5. DESIRED FEEDBACK

Though our pilot study found that p_f was greater than p_r for 74% of suites, we had hoped for a more substantial difference, since we can expect approximately 50% of suites to have $p_f > p_r$ by chance. We are considering other reduction methods that may produce a greater difference and welcome any suggestions regarding alternate reduction strategies. We also want to explore why the percentage varied with suite size and invite discussion of this topic.

6. CONCLUSION

Mutation testing can be too slow for practical use. One way to improve performance is to reduce the number of mutants to be evaluated. We proposed a new non-random mutant reduction method: evaluating only the faulty-file mutants, or mutants generated by altering files that are known to have contained many faults in the past. Our pilot study suggests that this method preferentially chooses mutants that will survive mutation testing. These mutants are interesting to developers because they indicate possible inadequacies in the test suite. The reduction technique also greatly decreases the amount of time required to evaluate the mutants, making mutation testing more practical. We plan to do a full study that confirms and extends these results.

7. REFERENCES

[1] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Trans. on Software Engineering*, 32(8):608–624, 2006.

[2] E. Arisholm and L. C. Briand. Predicting Fault-Prone Components in a Java Legacy System. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 8–17. ACM, 2006.

[3] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the Determination of Sufficient Mutant Operators for C. *Software Testing, Verification and Reliability*, 11(2):113–136, June 2001.

[4] M. Daran and P. Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. In *Proceedings of the Int. Symp. on Software Testing and Analysis*, pages 158–171, 1996.

[5] R. A. DeMillo, E. W. Krauser, and A. P. Mather. Compiler-Integrated Program Mutation. In *Proceedings of the International Computer Software and Applications Conference*, pages 351–356, 1991.

[6] N. E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *Trans. on Software Engineering*, 26(8):797–814, 2000.

[7] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *Transactions on Software Engineering*, 8(4):371–379, 1982.

[8] L. Inozemtseva. *Predicting Test Suite Effectiveness for Java Programs*. Master's thesis, University of Waterloo, 2012.

[9] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient Mutation Operators for Measuring Test Effectiveness. In *Proceedings of the Int. Conference on Software Engineering*, pages 351–360, 2008.

[10] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[11] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation Testing of Software Using a MIMD Computer. In *Proceedings of the Int. Conf. on Parallel Processing*, pages 257–266, 1992.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller. When Do Changes Induce Fixes? In *ACM SIGSOFT Software Engineering Notes*, pages 1–5. ACM, 2005.

[13] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation Analysis Using Mutant Schemata. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 139–148, 1993.

[14] W. E. Wong and A. P. Mathur. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[15] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. In *Proc. of the Workshop on Soft. Testing, Verif. and Analysis*, pages 152–158, 1988.

[16] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is Operator-Based Mutant Selection Superior to Random Mutant Selection? In *Proceedings of the Int. Conf. on Software Engineering*, pages 435–444, 2010.

[17] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression Mutation Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 331–341, 2012.