Mutation Testing Meets Approximate Computing

Milos Gligoric[1], Sarfraz Khurshid[1], Sasa Misailovic[2], and August Shi[2]
[1]The University of Texas at Austin, [2]University of Illinois at Urbana-Champaign
{gligoric, khurshid}@utexas.edu, {misailo, awshi2}@illinois.edu

***Abstract*—One of the most widely studied techniques in software testing research is mutation testing – a technique for evaluating the quality of test suites. Despite over four decades of academic advances in this technique, mutation testing has not found its way to mainstream development. The key issue with mutation testing is its high computational cost: it requires running the test suite against not just the program under test but against typically thousands of mutants, i.e., syntactic variants, of the program. Our key insight is that exciting advances in the upcoming, yet unrelated, area of approximate computing allow us to define a principled approach that provides the benefits of traditional mutation testing at a fraction of its usually large cost.**

This paper introduces the idea of a novel approach, named APPROXIMUT, that blends the power of mutation testing with the practicality of approximate computing. To demonstrate the potential of our approach, we present a concrete instantiation: rather than executing tests against each mutant on the exact program version, APPROXIMUT obtains an approximate test/program version by applying approximate transformations and runs tests against each mutant on the approximated version. Our initial goal is to (1) measure the correlation between mutation scores on the exact and approximate program versions, (2) evaluate the relation among mutation operators and approximate transformations, (3) discover the best way to approximate a test and a program, and (4) evaluate the benefits of APPROXIMUT. Our preliminary results show similar mutation scores on the exact and approximate program versions and uncovered a case when an approximated test was, to our surprise, better than the exact test.

I. INTRODUCTION

Mutation testing techniques evaluate the quality of a test suite by systematically inserting small syntactic changes into the code under test, e.g., replacing a '+' operator with '-', to generate *mutants* (the changed program versions), and then measuring the number of mutants killed by the given test suite [1], [13], [14]. A mutant is *killed* by the test suite if any test from the test suite fails when run on the mutant. The overall quality of a test suite – *mutation score* – is measured as the fraction of killed mutants.

Although valuable, mutation testing is costly due to a large number of mutants and tests. While researchers have explored various techniques to reduce the mutation testing cost, including mutant schemata, higher-order mutants, selective mutation, random selection, sampling mutation, etc. [13], [18], mutation testing is rarely used in practice because of the perceived costs.

We propose a novel approach, called APPROXIMUT, to reduce the mutation testing cost via approximate computing. *Approximate computing* is an emerging area of computer science that exposes sources of approximation at the computer system level and explicitly reasons about the tradeoff between accuracy and performance. Researchers have proposed various techniques for approximation in programming languages [6], [22], compilers [16], [20], [21], systems [3], [11], and hardware architectures [10], [15], [19]. For instance, approximate compilers automatically apply transformations that change the program semantics to trade off quality of results for better performance. These techniques exploit the inherent approximate nature of many applications (e.g., in domains such as image/multimedia processing, data mining, machine learning, and financial and engineering simulations).

The primary goal of APPROXIMUT is to reduce the cost of mutation testing for test suite evaluation, specifically the cost of computing (1) the mutation score for one test suite, or (2) the relation between mutation scores for a pair of test suites. We do not require the test suite to always be run using approximation; if the purpose of running the test suite is to find bugs in the program under test, the execution – which is against just one program and not many mutant programs – should indeed be non-approximate.

In our opinion, the specific context of mutation testing makes the application of approximate computing in testing appealing because computing a mutation score is inherently a heuristic computation, and its accuracy can be relaxed. As an illustration, consider two scenarios of using mutation testing:

- Computing mutation score of one test suite: the goal is to preserve mutation score – overall – across many mutant programs. The desired quality of the computed score is met whenever the mutation score is within a developer-specified threshold, even if approximation operators make (erroneously) some mutants killed or not killed.
- Comparing the mutation scores of two test suites: the goal is to preserve the relative mutation scores for the two suites. Even if the approximate mutation score of each test suite individually is significantly different from its non-approximate mutation score, as long as the relative approximate mutation scores are in the same order as the non-approximate mutation scores, we achieve the desired goal of comparing two test suites correctly.

APPROXIMUT starts by approximating tests and the program under test by applying *approximate transformations* (e.g., transform a loop to execute fewer iterations). Then, APPROXIMUT determines expected values in each test for the approximate program version. Finally, APPROXIMUT measures the mutation score by treating the approximate program as the exact program.

Our approach opens a number of research questions, including (1) what is the correlation between mutation scores on

```
1  // MathArraysTest.java
2  void testUnique() {
3    double[] x = {0, 9, 3, 0, 11,
4                  7, 3, 5, −1, −2};
5    double[] values = { 11, 9, 7, 5, 3,
6                        0, -1, -2 };
7    assertArrayEquals(values,
8        MathArrays.unique(x), 0);
9  }
10 // MathArrays.java
11 static double[] unique(double[] data) {
12   TreeSet<Double> values =
13       new TreeSet<>();
14   for (int i = 0; i < data.length; i++) {
15
16     values.add(data[i]);
17   }
18   int count = values.size();
19   double[] out = new double[count];
20   Iterator<Double> iterator =
21       values.descendingIterator();
22   int i = 0;
23   while (iterator.hasNext()) {
24
25     out[i++] = iterator.next();
26   }
27   return out;
28 }
```

(a)

```
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
                7, 3, 5, −1, −2};
  double[] values = {11, 9, 7, 5, 3,
                     0, 0, 0};
  assertArrayEquals(values,
      MathArrays.unique(x), 0);
}
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
      new TreeSet<>();
  for (int i = 0; i < data.length-1; i++) {

    values.add(data[i]);
  }
  int count = values.size();
  double[] out = new double[count+1];
  Iterator<Double> iterator =
      values.descendingIterator();
  int i = 0;
  while (iterator.hasNext()) {
    if (i == data.length / 2 + 1) break;
    out[i++] = iterator.next();
  }
  return out;
}
```

(b)

```
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
                7, 3, 5, −1, −2};
  double[] values = {11, 3,
                     0, -1};
  assertArrayEquals(values,
      MathArrays.unique(x), 0);
}
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
      new TreeSet<>();
  for (int i = 1; i < data.length; i++) {
    if (i%2 != 0) continue;
    values.add(data[i]);
  }
  int count = values.size();
  double[] out = new double[count];
  Iterator<Double> iterator =
      values.descendingIterator();
  int i = 0;
  while (iterator.hasNext()) {

    out[i++] = iterator.next();
  }
  return out;
}
```

(c)

Fig. 1: Example test from Apache Math project [2]. Loop perforation is highlighted in bright and mutants in dark blue. (a) The exact code and test. (b) The exact code kills the mutant, but the approximate version does not. (c) The approximate code kills the mutant, but the exact version does not.

the exact and approximate program versions, (2) what is the relation between mutation operators and approximate transformations, (3) what approximations maximize the performance of tests and programs, and (4) what is the speedup that can be obtained by APPROXIMUT.

In this paper, we only tackle the first question by measuring mutation scores on several approximate program versions for a test from the Apache Commons Math project [2]. Our evaluation shows that the mutation scores of the approximate versions are similar to the mutation score of the exact program.

We believe that approximate computing holds a key to making software testing more effective. Indeed, testing, by definition, is a form of approximation – specifically, under-approximation in the sense of verification. Approximate computing provides a principled way to methodically introduce new, well-defined approximations into testing techniques to increase their efficacy, thereby allowing software testing to play an even more vital role in developing more reliable software systems.

II. PRELIMINARY STUDY

This section describes how APPROXIMUT approximates mutation score computation by using a specific approximate transformation: loop perforation.

A. Example

Exact program. Figure 1a shows a code snippet from the Apache Commons Math (SHA: d9e43edd) [2]. The method `unique` removes duplicate values from the input array (Lines 12-17) and sorts the values in descending order (Lines 20-26). The method `testUnique` invokes `unique` with the argument `x` and checks whether the actual result matches the expected values from the array `values`.

Loop perforation. Loop perforation is a compiler-level transformation [16], [24], which transforms loops such as `for (int i = 0; i < n; i++) {...}` to execute only a subset of its iterations. We consider two forms of perforation:

- *Interleaving perforation*: skipping intermediate loop iterations. It skips every 2nd or 4th iteration, or executes every 4th iteration. We applied this transformation to the first loop in the subject (Line 15 in Figure 1c).
- *Truncating perforation*: skipping the last quarter, half, or three quarters of iterations. We applied this transformation to the second loop in the subject (Line 24 in Figure 1b).

Loop perforation typically makes computations run faster by doing less work, but also produces different, typically less accurate results. The fraction of iterations to execute is a parameter that controls the tradeoff space for the computation.

Approximations. Figures 1b and 1c present two approximate versions of this computation that apply loop perforation to speed up mutation score calculation. The implementation of perforation is highlighted with bright (blue) color. The perforations cause the first loop to add only a subset of the elements in the ordered set `values` and the second loop to read only the first few elements `values` (in descending order).

After applying perforation to the function `unique`, APPROXIMUT also needs to repair the test to reflect the change in the results of the tested function, e.g., using the approach from ReAssert [8]. For the example 1b, the approximate function computes the ordered list of the first six elements of the array correctly and keeps the remaining two as zeros. The

test changes the constants for these two array elements (highlighted). For the example 1c, the approximate result contains a subset of odd-positioned elements, including the maximum element, but the resulting array is smaller (highlighted).

Performance. Profiling reveals that the first loop (Line 15) consumes over 68% of the run time and the second loop (Line 24) consumes less than 7% of the run time. Therefore, perforation of the first loop, which inserts elements in a `TreeSet` – an $O(n \log n)$ operation – can contribute more to the run time savings than the perforation of the second loop, which does a linear scan over the tree elements.

B. Mutation Score Optimization

Optimization statement. We navigate the tradeoff space induced by three transformations: interleaving loop perforation, truncating loop perforation, and *input reduction* (which removes a part of the input array from the test – e.g., one half or one quarter of the elements). The quality metric measures the effectiveness of a test suite. Specifically, APPROXIMUT computes *mutation score*, as the number of killed mutants divided by the total number of mutants. Our search is looking for the versions of the test suite that yield maximum performance savings while computing the mutation score that is within a given percent of the mutation score of the exact test suite. Such a defined set of transformations, a quality metric and a quality degradation are sufficient for automatic exploration of the quality/performance tradeoff space [16], [24].

Mutation score. We obtained mutants and computed mutation score by running the PIT mutation testing tool[1] on the subject program from Figure 1a. We configured PIT to use all mutation operators it supports; the operators modify constants, conditional statements, and method calls. PIT generated a total of 22 mutants for the subject program. The number of first-order mutants (those where only one program location is changed) is the same for the exact version of program and for all approximate programs, i.e., we do not mutate code that is inserted by approximate transformations.

Specifically, for our subject program, we (1) apply one approximate transformation at a time, (2) execute the test to obtain the expected value, (3) update the expected value in the approximate test, and (4) run PIT with the approximate program version to compute mutation score.

Results. Table I shows the mutation score values for the exact and the approximate program versions. The first column shows the approximate transformation type that was used, the second column specifies the instance of the approximate transformation, and the third column shows the mutation score. ("Exact version" row denotes the exact program version.)

The results show that the mutation scores computed on approximate programs are only a few percentage points from the exact mutation score. For this subject, the approximate programs obtained by "Input Summarization" have the most similar mutation scores to the exact program.

[1]PIT is available at http://pitest.org

TABLE I: Mutation score for the exact and various approximate program versions.

Approximate Transformation	Instance	Mutation Score [%]
Exact version	-	95
Interleaving perforation (loop on Line 15)	skip 2nd iter.	100
	skip 4th iter.	95
	execute 4th iter.	100
Truncating perforation (Loop on Line 24)	skip last 1/4 iters.	91
	skip half iters.	91
	skip last 3/4 iters.	91
Input summarization	remove last 1/4 iters.	95
	remove half iters.	100
	remove last 3/4 iters.	95

To our surprise, we identified several cases (e.g., "Input Summarization" with "Remove Half") that obtain higher mutation scores than the exact test. This is especially interesting in the case of "Input Summarization" because the approximation does not alter the exact control flow of the program but only provides a different input. In other words, by approximating test inputs, we could obtain a test suite with better fault-detection capability (as measured by a mutation score).

III. FUTURE DIRECTIONS

Approximating tests can lead to improved performance of the overall testing, but it also opens up a number of research questions. We discuss below several of them.

Can mutation testing be optimized by computing the mutation score on an approximate version of the program? To answer this question, we need to find the common relations between mutation scores for the exact program/test and approximate program/test. We know that a mutant killed by the exact program does not have to be killed by the approximate program, and vice versa. We illustrated these two cases in Section II, Figures 1b (the exact test kills the mutant) and 1c (the exact test does not kill the mutant), respectively.

What is the relation between mutants and approximate transformations? For instance, modification of the loop induction variable increment from `i=i+1` to `i=i+2` can be a result of both program approximation and program mutation. Our insight is that certain approximate transformations can be valid mutation operators. However, only some mutation operators can be seen as approximate transformations.

We plan to study the relation between mutation operators and approximate transformations. To the best of our knowledge, no prior work has explored this relation. We plan to evaluate if some approximate transformations subsume existing mutation operators (or vice versa) and what mutation operators may impact program efficiency. We will also compare this set of mutants with the mutants obtained through selective mutation [17]. Finally, we plan to investigate the relation of mutants that make multiple code modifications [13] (i.e., higher-order mutants) with approximate transformations.

What are the mutation operators that are appropriate for approximate programs? We plan to develop new mutation

operators appropriate for approximate programs. For example, rather than mutating constant integer values, which is one of the most common operators, we plan to mutate floating point numbers. Furthermore, we can mutate the distribution function (e.g., uniform to Bernoulli), which generates the input data used to compute parts of the output. Toward this goal, selective mutation [4], [17], [18], [23] is a promising research direction that can be applied to find mutation operators that subsume the others and discover relations between traditional mutation operators and mutation operators for approximate programs.

Can we use approximation to speed up test execution? The preliminary study considered only approximating mutation score computation. Once the developer has enough confidence in the test suite, he or she runs the original (exact) test suite. However, we could also use approximated tests (e.g., like those in Figures 1b and 1c during test execution. The main challenge, however, is ensuring that the approximated test is likely to reveal bugs that the original test would have revealed. One conservative approach is to find transformations that ensure the approximated test will reveal a bug if the original test reveals it, but it may signal false alarms even when the original test does not. Another more liberal approach is to specify relaxations of the test oracles (but such that they preserve key safety properties). For instance, the list of unique elements should contain all the values, but only a *majority of them* need to be ordered.

Which applications are amenable to approximate mutation testing? We anticipate that the approximate mutation testing is particularly well-suited for application domains that have an inherent notion of quality of results. However, in some scenarios, approximate transformations can be used on arbitrary programs, e.g., if an approximation is used as a (general) mutation operator.

IV. Related Work

We propose the first technique that applies mutation testing and approximate computing in tandem. Previous work on optimizing mutation testing focused primarily on *selective mutation* [17], which reduces the number of mutants to run by applying a subset of mutation operators that *subsume* the other operators, or by running a select subset of the pool of candidate mutants. In contrast, ApproxiMut optimizes *the execution of individual tests* regardless of the set of used mutation operators. *Predictive mutation* [26] computes the mutation score without running tests on mutants; however, it allows lesser control over mutation score accuracy.

The spirit of approximation in testing is common. For example, *test-suite reduction* [25] can remove the entire tests from a test suite. In contrast, ApproxiMut reduces the execution of each test while keeping all tests in the test suite.

Testability transformations introduced the use of program transformations for improved testing [12]. Recent follow-up work used standard transformations, e.g., compiler optimizations, as well as ad hoc, non-semantics-preserving transformations in the context of symbolic execution [5], [7], [9].

V. Conclusion

This paper advocates the integration of testing – the most widely used method for validating quality of software – and automated approximation – a promising new approach for developing and optimizing an increasingly important class of programs. In particular, the insights at the heart of approximation can be leveraged to introduce a fresh approach for rethinking mutation testing techniques, which already have a rich history, and making them even more effective and efficient.

Acknowledgments

We thank Farah Hariri and Owolabi Legunsen for useful feedback. This research was partially supported by NSF grants CCF-1409423, CCF-1421503, CCF-1566363, CCF-1629431, CCF-1319688, and CNS-1239498.

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
[2] Commons Math. https://github.com/apache/commons-math.git.
[3] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.
[4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *STVR*, 11(2), 2001.
[5] C. Cadar. Targeted program transformations for symbolic execution. In *ESEC/FSE*, 2015.
[6] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA,* 2013.
[7] H. Converse, O. Olivo, and S. Khurshid. Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution. In *ICST*, 2017. (To appear).
[8] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *ASE*, 2009.
[9] S. Dong, O. Olivo, L. Zhang, and S. Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *ISSRE,* 2015.
[10] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.
[11] I. Goiri, R. Bianchini, S. Nagarakatte, and T. Nguyen. ApproxHadoop: Bringing approximations to MapReduce frameworks. In *ASPLOS*, 2015.
[12] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *TSE*, 30(1), 2004.
[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5), 2011.
[14] R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? 2014.
[15] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *DATE*, 2010.
[16] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *ICSE*, 2010.
[17] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *ICSE*, 1993.
[18] A. J. Offutt and R. H. Untch. *Mutation 2000: Uniting the Orthogonal*. Springer US, 2001.
[19] K. Palem. Energy aware computing through probabilistic switching: a study of limits. *Transactions on Computers*, 54(9), 2005.
[20] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, 2006.
[21] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.
[22] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.
[23] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE*, 2008.
[24] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.
[25] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2), 2012.
[26] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang. Predictive mutation testing. In *ISSTA*, 2016.