Mutation Testing Meets Approximate Computing

Milos Gligoric[1], Sarfraz Khurshid[1], Sasa Misailovic[2], **August Shi[2]**

ICSE NIER 2017

Buenos Aires, Argentina

May 24, 2017

1
THE UNIVERSITY OF TEXAS AT AUSTIN

NSF
CCF-1409423, CCF-1421503, CCF-1566363, CCF-1629431, CCF-1319688, CNS-1239498

2
ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Mutation Testing

- **Goal:** Evaluate quality of test suites
- **How:** Apply transformations (mutation operators) on the code and run tests to see if they can detect the changes
- **Example:**

```
x = x + 1
```

Mutation Testing

- **Goal:** Evaluate quality of test suites
- **How:** Apply transformations (mutation operators) on the code and run tests to see if they can detect the changes
- **Example:**

```
x = x + 2
```

- **Problems:**
  - Evaluation of quality limited by mutation operators
  - Too slow

Approximate Computing

- **Goal:** Improve performance of code
- **How:** Apply transformations that may lead to (slightly) inaccurate results
- **Example:**

```
for (i = 0; i < n; i = i + 1)
```

Approximate Computing

- **Goal:** Improve performance of code
- **How:** Apply transformations that may lead to (slightly) inaccurate results
- **Example:**

```
for (i = 0; i < n; i = i + 2)
```

- **Problems:**
  - Not sure where in exact code to apply approximations
  - Unclear how to check quality of tests on already approximate code

How can
Mutation Testing and
Approximate Computing
improve one another?

Improving One Another

- Approximate computing to provide new mutation operators for **evaluating quality of tests**
- Approximate computing to **improve speed** of mutation testing
- Mutation testing to **point out opportunities** for applying approximations on exact code
- Mutation testing to **evaluate quality of tests** on (already) approximate code

Example Code: Commons-Math

```java
// MathArrays.java
static double[] unique(double[] data) {
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

✔ **Test Passes**

Mutant: Constant Replacement

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 0; i < data.length; i++) {
    values.add(data[i]);
  }
  int count = values.size();
  double[] out = new double[count];
  Iterator<Double> iterator =
    values.descendingIterator();
  int i = 1;
  while (iterator.hasNext()) {
    out[i++] = iterator.next();
  }
  return out;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

Test Fails

Mutant Killed

Replace 0 with **1**

Mutant: Constant Replacement

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 1; i < data.length; i++) {
    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;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

Test Passes

Mutant Survived

Replace 0 with **1**

Approximate: Loop Perforation

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 0; i < data.length; i+=2) {
    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;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

Test Fails

Skip every other iteration

68% of runtime is in this loop

Modify assertion

Approximate: Loop Perforation

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 0; i < data.length; i+=2)
    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;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArraySubset(values,
    MathArrays.unique(x), 0);
}
```

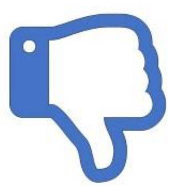
Test Passes

Approximation is Acceptable

Skip every other iteration

68% of runtime is in this loop

Comparison of Transformation Results

	Mutation Testing	Approximate Computing
Failing Test		
Passing Test		

Comparison of Transformation Results

	Mutation Testing	Approximate Computing
Failing Test		
Passing Test		

Approx. Transformation as Operator

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 0; i < data.length; i++)
    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;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

Approx. Transformation as Operator

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 0; i < data.length; i+=2)
    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;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

Test Fails

Mutant Killed

Replace `i++` with **i+=2**

Perforates loop

Questions for Approx. Operators

- Do killed approximate mutants indicate different strengths? Do surviving approximate mutants indicate new weaknesses in the test suite?

- How do mutants generated by approximate computing differ from traditional mutants?

- Are approximate mutants faster than traditional mutants?

Mutants Find Approx. Opportunities

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 0; i < data.length; i++)
    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;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

Mutants Find Approx. Opportunities

```java
// MathArrays.java
static double[] unique(double[] data) {
  TreeSet<Double> values =
    new TreeSet<>();
  for (int i = 1; i < data.length; i++)
    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;
}
```

```java
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
    7, 3, 5, -1, -2};
  double[] values = {11, 9, 7,
    5, 3, 0, -1, -2};
  assertArrayEquals(values,
    MathArrays.unique(x), 0);
}
```

Test Passes

Approximable?

Replace 0 with **1**

Questions for Approx. Opportunities

- How can we classify surviving mutants?
  Are they good for approximate computing?
- What approximations are applicable for which surviving mutants?
- How can we tailor mutants for the purpose of finding approximate computing opportunities?

Improving One Another

- Approximate computing to provide new mutation operators for **evaluating quality of tests** ✓
- Approximate computing to **improve speed** of mutation testing
- Mutation testing to **point out opportunities** for applying approximations on exact code ✓
- Mutation testing to **evaluate quality of tests** on (already) approximate code

More in paper!

Conclusions

- Approximate computing can provide new mutation operators
- Mutation testing can show opportunities for approximate computing on exact code
- There is so much more we can do (More directions in the paper)

August Shi: awshi2@illinois.edu

BACKUP

Mutating Approximate Code?

```
    // 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]);
  }
  return values.toArray();
}
```

```
// MathArraysTest.java
void testUnique() {
  double[] x = {0, 9, 3, 0, 11,
                7, 3, 5, -1, -2}
    double[] values = {11, 9, 7,
                5, 3, 0, -1, -2};
    assertArrayEquals(values,
                MathArrays.unique(x), 0);
}
```

✔ **Test Passes**

Replace 0 with **1**

Mutation testing approximate code?

Approximate Transformation	Instance	% Mutants Killed
Exact Version	N/A	95
Loop Perforation (Line 04)	Skip every 2nd iteration	100
	Skip every 4th iteration	95
	Only execute every 4th iteration	100

What precisely do these changes in percentages mean?

Directions for Research

- If approximate version is proxy of exact version, is mutation score of approximate version also proxy of mutation score of exact version?
  - Do I get same confidence in quality of tests at cheaper cost?
- If so, what are the exact conditions where they are good proxies?

Approximate Code to Speed up Testing?

```java
    // MathArrays.java
01. static double[] unique(double[] data) {
02.     TreeSet values =
03.         new TreeSet<>();
04.     for (int i = 0;
05.         i < data.length; i++) {
06.         if (i%2 != 0) continue;
07.         values.add(data[i]);
08.     }
09.     int count = values.size();
10.     double[] out = new double[count];
11.     Iterator iterator =
12.          values.descendingIterator();
13.      int i = 0;
14.      while (iterator.hasNext()) {
15.
16.         out[i++] = iterator.next();
17.     }
19.     return out;
20. }
```

```java
// 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);
}
```

Loop (line 04) takes 68% of runtime

Perforation can cut time in half

Approximate Code to Speed up Testing?

```java
// MathArrays.java
static double[] unique(double[] data) {
    TreeSet values =
        new TreeSet<>();
    for (int i = 0;
        i < data.length; i++) {
        if (i%2 != 0) continue;
        values.add(data[i]);
    }
    int count = values.size()
    double[] out = new double[count];
    Iterator iterator =
         values.descendingIterator();
     int i = 0;
     while (iterator.hasNext()) {

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

```java
// MathArraysTest.java
void testUnique() {
    double[] x = {0, 9, 3, 0, 11,
        7, 3, 5, -1, -2}
    double[] values = {11, 9, 7,
        5, 3, 0, -1, -2};
    assertApproxEquals(values,
        MathArrays.unique(x), 0
        0.8);
}
```

Introduce new assertions?

Example: Mutation Testing (SURVIVED)

```java
// MathArrays.java
static double[] unique(double[] data) {
    TreeSet values =
        new TreeSet<>();
    for (int i = 0;
        i < data.length; i++) {

        values.add(data[i]);
    }
    int count = values.size()
    double[] out = new double[count];
    …
    return out;
}
```

Original

Constant Replacement

```java
// MathArrays.java
static double[] unique(double[] data) {
    TreeSet values =
        new TreeSet<>();
    for (int i = 1;
        i < data.length; i++) {

        values.add(data[i]);
    }
    int count = values.size()
    double[] out = new double[count];
    …
    return out;
}
```

Mutant

```java
// MathArraysTest.java
void testUnique() {
    double[] x = {0, 9, 3, 0, 11,
        7, 3, 5, -1, -2}
    double[] values = {11, 9, 7, 5, 3,
        0, -1, -2};
    assertArrayEquals(values,
        MathArrays.unique(x), 0);
}
```

Test passes on original
Test passes on mutant => Mutant SURVIVED

Example: Mutation Testing (KILLED)

```java
// MathArrays.java
static double[] unique(double[] data) {
    TreeSet values =
        new TreeSet<>();
    for (int i = 0;
        i < data.length; i++) {
        values.add(data[i]);
    }
    int count = values.size()
    double[] out = new double[count];
    …
    return out;
}
```

Original

Boundary Mutator

```java
// MathArrays.java
static double[] unique(double[] data) {
    TreeSet values =
        new TreeSet<>();
    for (int i = 0;
        i <= data.length; i++) {
        values.add(data[i]);
    }
    int count = values.size()
    double[] out = new double[count];
    …
    return out;
}
```

Mutant

```java
// MathArraysTest.java
void testUnique() {
    double[] x = {0, 9, 3, 0, 11,
        7, 3, 5, -1, -2}
    double[] values = {11, 9, 7, 5, 3,
        0, -1, -2};
    assertArrayEquals(values,
        MathArrays.unique(x), 0);
}
```

Test passes on original
Test passes on mutant => Mutant KILLED

Example: Loop Perforation

```java
// MathArrays.java
static double[] unique(double[] data) {
    TreeSet values =
        new TreeSet<>();
    for (int i = 0;
        i < data.length; i++) {

        values.add(data[i]);
    }
    int count = values.size()
    double[] out = new double[count];
    …
    return out;
}
```

Original

Skip Every Other Iteration

```java
// MathArrays.java
static double[] unique(double[] data) {
    TreeSet values =
        new TreeSet<>();
    for (int i = 0;
        i < data.length; i+=2  {

        values.add(data[i]);
    }
    int count = values.size()
    double[] out = new double[count];
    …
    return out;
}
```

Mutant