CS 4530: Fundamentals of Software Engineering
Module 3.2: Code Coverage & Mutation Testing

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

At the end of this lesson, you should be able to

- Explain what code coverage is, and how different measures differ, including statements, branches, functions, and lines
- Explain the benefits of mutation testing

When have I written enough tests?

Why do we test?

- Test Driven Development
  - If we start with tests, passing the tests lets us know we're done meeting the conditions of satisfaction
- Acceptance Testing
  - Does the customer agree we've met the conditions of satisfaction?
- Regression Testing
  - Did something change since some previous version?
  - Prevent bugs from (re-)entering during maintenance.
  - "Good" test suite detects bugs that (inevitably) get added as software is developed over time: tests are for *the future*

When Have I Written Enough Tests?

- When I've tested the valid inputs
- **When I've tested all the code**
- **When the tests will catch bugs**

Code Coverage

- The industry standard answer for “have I written enough tests”
- Measures “how much of your code” is exercised by your tests
- If none of your test even *execute* a piece of code, it’s not being tested!

Code Coverage: Line or Statement Coverage

- *Line* and *Statement* coverage: coarsest measure.
- Testing x = 0 exercises lines 1 and 2
- Testing x = 10 exercises lines 1, 4, 5, and 6.

```
1 if (x === 0) {
2  return 3;
3 }
4 const y = x > 4 ? 2 : 3;
5 const z = x % 2 == 0 ? 1 : 2;
6 return x / (y - z);
```

Code Coverage: Branch Coverage

- *Branch* coverage: most widely used in industry.
- Testing with x > 4 and x <= 4 necessary to handle both branches on line 4.
- Testing with odd and even numbers necessary to handle both branches on line 5.
- The values -2, 0, 1, and 10 get full branch coverage.

```
1 if (x === 0) {
2  return 3;
3 }
4 const y = x > 4 ? 2 : 3;
5 const z = x % 2 == 0 ? 1 : 2;
6 return x / (y - z);
```

Code Coverage: Path Coverage

- The values -2, 0, 1, and 10 get full branch coverage…
- …but 5 causes line 6 to divide by zero!
  - In JavaScript/TypeScript, this doesn't cause an exception, there's a number called "NaN" for "not a number"
- *Path* coverage covers all combinations of branches; it is infeasible in practice.

```
1 if (x === 0) {
2   return 3;
3 }
4 const y = x > 4 ? 2 : 3;
5 const z = x % 2 == 0 ? 1 : 2;
6 return x / (y - z);
```

Code Coverage: Example 1

What tests will get line/branch/path coverage here?

```
/** Returns an array that repeats "hello"
 * @param numHellos - number of “hello”s to return, must be an integer >= 0
 */
function helloNTimes(numHellos: number): string[] {
  const arr: string[] = [];
  for (let i = numHellos; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

Code Coverage: Example 2

What tests will get line/branch/path coverage here?

```
const zHelloInput = z.int().gte(0);

/** Returns an array that repeats "hello"
 * @param numHellos - number of times to say “hello”
 * @throws if the input is not an integer >= 0
 */
function helloNTimes(numHellos: unknown): string[] {
  const parseResult = zHelloInput.safeParse(numHellos);
  if (!parseResult.success) throw new Error("Invalid input");
  const arr: string[] = [];
  for (let i = parseResult.data; i !== 0; i--) { arr.push("hello"); }
  return arr;
}
```

Code Coverage: Example 2

What tests will get line/branch/path coverage here?

```
type zAuth = z.object({ username: z.string(), password: z.string() })

function useAuth(x: unknown) {
  const auth = zAuth.safeParse(x);
  if (!auth.error) {
    // write the code you care about here!
  }
}
```

Code Coverage: More Examples

- Some of our intuition about “edge cases” can be understood in terms of trying to imagine inputs that will provide code coverage for an unknown function.
  - BUT: code coverage won’t tell you to check 0 specifically if there’s a branch that checks x >= 0
  - Thinking about the “edge case” of a branch is a good move
- What inputs should we start with if:
  - We’re taking numbers that are supposed to represent a 5-digit zip code?
  - We’re taking strings that are supposed to represent a 5-digit zip code?
  - We’re taking an array of strings that represent people’s names?

Code Coverage in Vitest

- Tools like vitest makes it easy to check code coverage

```
  calculator/add
    ✓ should return a number when parameters are passed to `add()`
    ✓ should return sum of `2` when 1 + 1 is passed to `add()`

  calculator/subtract
    ✓ should return a number when parameters are passed to `subtract()`
    ✓ should return sum of `1` when 2 - 1 is passed to `subtract()`


  4 passing (4ms)

-------------|---------|----------|---------|---------|-------------------
File         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------|---------|----------|---------|---------|-------------------
All files    |     100 |      100 |     100 |     100 |
 Add.ts      |     100 |      100 |     100 |     100 |
 Subtract.ts |     100 |      100 |     100 |     100 |
-------------|---------|----------|---------|---------|-------------------
```

Code Coverage: Review

- Total code coverage — by any metric — does not mean no bugs
  - Running code doesn't mean checking that it's doing the right thing! (important to Assemble/Act/Assess)
  - "Coverage is no substitute for thought" — Rob Pike
- Coverage checking can be invaluable at identifying when you *think* you're testing something but you're not, which is a real problem in practice.
  - Test-Driven Development also valuable for this problem: it's important that tests switch from failing to succeeding *when you expect them to*

Adversarial Testing

- It can be helpful to think of testing as a game in which you play against an adversary.
- Your adversary plays by producing multiple versions of code that you agree is buggy, and multiple versions of code you agree is correct.
- Your win if your tests catch all the buggy code, and pass all the correct code.

Adversarial Testing

Original code (correct)

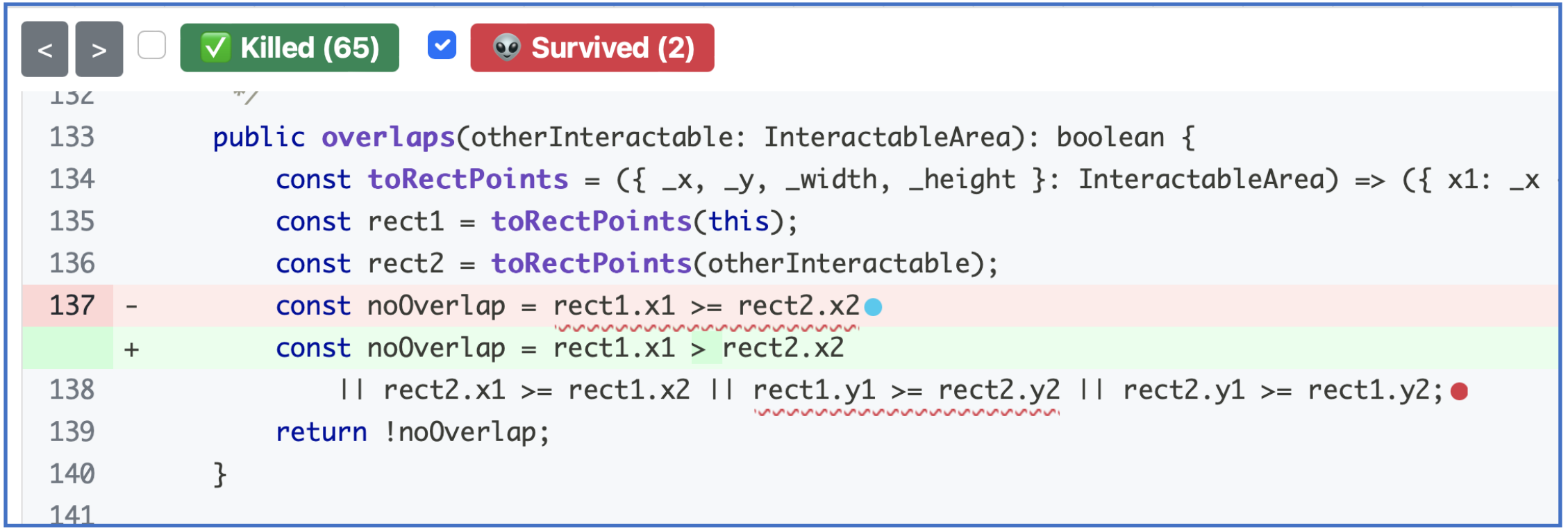
```
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number) {
    return list.find((item) => item >= target);
}
```

Mutated code (buggy)

```
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number) {
    return list.find((item) => item > target);
}
```

Adversarial Testing

Stryker is a *mutation tester* for JavaScript — an automated adversary!



Remedy: you need to devise tests that distinguish the original code from the mutants

- Devise a test that your original code will pass, but the mutant will fail.

A tiny example

Imagine that this is the code to be tested

```
// find the first item in the list that is
// greater than or equal to the target.
export default function search(list:number[], target:number)
{
    return list.find((item) => item >= target);
}
```

and we have written some tests.

Stryker report for this test

```
[Survived] EqualityOperator
src/for-midterm/adrian.ts:4:32
-       return list.find((item) => item >= target);
+       return list.find((item) => item > target);
Tests ran:
    search should return the first item in the list that is greater
than or equal to the target
```

The location of the mutation

```
[Survived] ConditionalExpression
src/for-midterm/adrian.ts:4:32
-       return list.find((item) => item >= target);
+       return list.find((item) => true);
Tests ran:
    search should return the first item in the list that is greater
than or equal to the target
```

The tests that were run on this piece of code

Let's look at the second one:

```
[Survived] ConditionalExpression
src/for-midterm/adrian.ts:4:32
-         return list.find((item) => item >= target);
+         return list.find((item) => true);
```

- How does this mutant behave differently from the original?
- Answer: This mutant always returns the first element of the list
- Remedy: Add a test that searches for something that is NOT the first element in the input?

Here's one test that will cause the mutant to be killed.

```
test("should return the second element of the list", () => {
  expect(search([5, 7, 9], 6)).toBe(7);
});
```

What about the other mutant?

```
[Survived] EqualityOperator
src/for-midterm/adrian.ts:4:32
-       return list.find((item) => item >= target);
+       return list.find((item) => item > target);
```

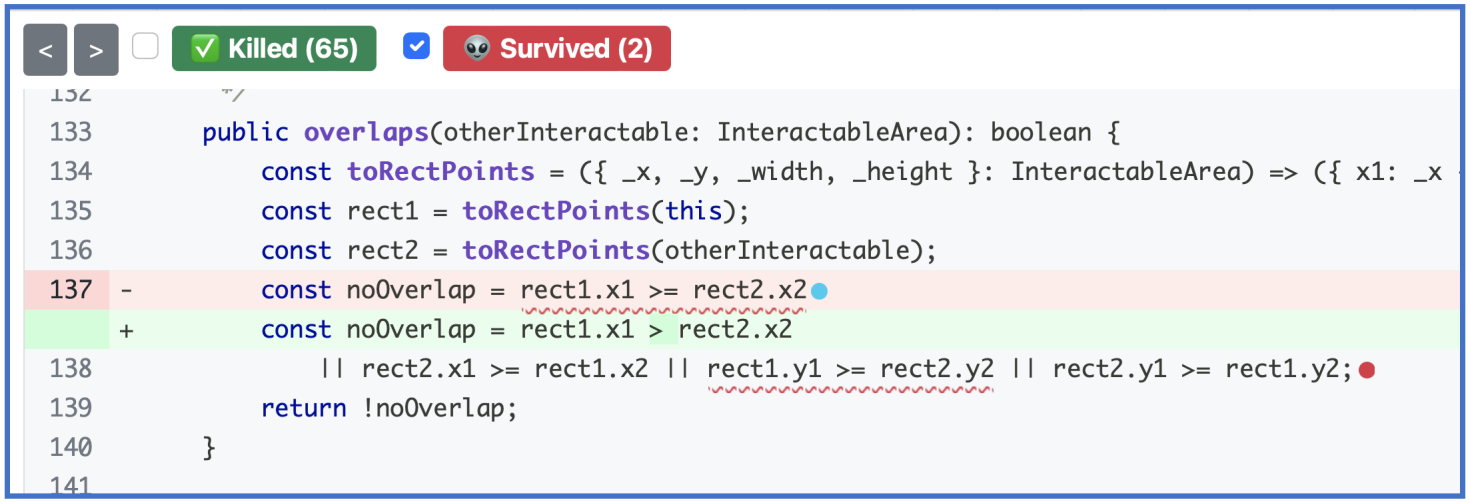
- How does this mutant behave differently from the original?
- Answer: This mutant returns the first element of the list that is larger than the original
- Remedy: Add a test where the input list included an “equal” item before a larger item

Here's one test that will catch that mutant

```
test("try target that is equal to some item in the list", () => {
    expect(search([5, 7, 9], 7)).toBe(7);
  });
```

Use Mutation Analysis While Writing Tests

- When you feel “done” writing tests, run a mutation analysis
- Inspect undetected mutants, and try to write tests that will make those mutants fail.



Detailed mutation report for “overlaps” method - two mutants were not detected!

Adversarial Testing and Overspecification

Stryker is a *mutation tester* for JavaScript — an automated adversary!

Sometimes it loses the game because mutants aren't bugs.

```
62        public static fromMapObject(mapObject: ITiledMapObject, broadcas
63            const { name, width, height } = mapObject;
64            if (!width || !height) {
65  -             throw new Error(`Malformed viewing area ${name}`);
    +             throw new Error(``);
66            }
67            const rect: BoundingBox = { x: mapObject.x, y: mapObject.y, w
68            return new ConversationArea({ id: name, occupantsByID: [] },
69        }
```

Are mutants a Valid Substitute for Real Faults? Probably yes.

- Do mutants really represent real bugs?
- Researchers have studied the question of whether a test suite that finds more mutants also finds more real faults
- Conclusion: For the 357 real faults studied, yes
- This work has been replicated in many other contexts, including with real faults from student code

Are Mutants a Valid Substitute for Real Faults in Software Testing?

René Just†, Darioush Jalali†, Laura Inozemtseva*, Michael D. Ernst†, Reid Holmes*, and Gordon Fraser‡
†University of Washington, Seattle, WA, USA, {rjust, darioush, mernst}@cs.washington.edu
*University of Waterloo, Waterloo, ON, Canada, {lminozem, rtholmes}@uwaterloo.ca
‡University of Sheffield, Sheffield, UK, gordon.fraser@sheffield.ac.uk

ABSTRACT

A good test suite is one that detects real faults. Because the set of faults in a program is usually unknowable, this definition is not useful to practitioners who are creating test suites, nor to researchers who are creating and evaluating tools that generate test suites. In place of real faults, testing research often uses mutants, which are artificial faults — each one a simple syntactic variation — that are systematically seeded throughout the program under test. Mutation analysis is appealing because large numbers of mutants can be automatically-generated and used to compensate for low quantities or the absence of known real faults.

Unfortunately, there is little experimental evidence to support the use of mutants as a replacement for real faults. This paper investigates whether mutants are indeed a valid substitute for real faults, i.e., whether a test suite's ability to detect mutants is correlated with its ability to detect real faults that developers have fixed. Unlike prior studies, these investigations also explicitly consider the confounding effects of code coverage on the mutant detection rate.

Our experiments used 357 real faults in 5 open-source applications that comprise a total of 321,000 lines of code. Furthermore, our experiments used both developer-written and automatically-generated test suites. The results show a statistically significant correlation between mutant detection and real fault detection, independently of code coverage. The results also give concrete suggestions on how to improve mutation analysis and reveal some inherent limitations.

Categories and Subject Descriptors

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

General Terms

Experimentation, Measurement

Keywords

Test effectiveness, real faults, mutation analysis, code coverage

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635929

1. INTRODUCTION

Both industrial software developers and software engineering researchers are interested in measuring test suite effectiveness. While developers want to know whether their test suites have a good chance of detecting faults, researchers want to be able to compare different testing or debugging techniques. Ideally, one would directly measure the number of faults a test suite can detect in a program. Unfortunately, the faults in a program are unknown a priori, so a proxy measurement must be used instead.

A well-established proxy measurement for test suite effectiveness in testing research is the *mutation score*, which measures a test suite's ability to distinguish a program under test, the *original version*, from many small syntactic variations, called *mutants*. Specifically, the mutation score is the percentage of mutants that a test suite can distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, using well-defined *mutation operators*. Examples of such mutation operators are replacing arithmetic or relational operators, modifying branch conditions, or deleting statements (cf. [18]).

Mutation analysis is often used in software testing and debugging research. More concretely, it is commonly used in the following use cases (e.g., [3, 13, 18, 19, 35, 37–39]):

Test suite evaluation The most common use of mutation analysis is to evaluate and compare (generated) test suites. Generally, a test suite that has a higher mutation score is assumed to detect more real faults than a test suite that has a lower mutation score.

Test suite selection Suppose two unrelated test suites T_1 and T_2 exist that have the same mutation score and $|T_1| < |T_2|$. In the context of test suite selection, T_1 is a preferable test suite as it has fewer tests than T_2 but the same mutation score.

Test suite minimization A mutation-based test suite minimization approach reduces a test suite T to $T \setminus \{t\}$ for every test $t \in T$ for which removing t does not decrease the mutation score of T.

Test suite generation A mutation-based test generation (or augmentation) approach aims at generating a test suite with a high mutation score. In this context, a test generation approach augments a test suite T with a test t only if t increases the mutation score of T.

Fault localization A fault localization technique that precisely identifies the root cause of an artificial fault, i.e., the mutated code location, is assumed to also be effective for real faults.

These uses of mutation analysis rely on the assumption that mutants are a valid substitute for real faults. Unfortunately, there is little experimental evidence supporting this assumption, as discussed in greater detail in Section 4. To the best of our knowledge, only three previous studies have explored the relationship between mutants and

Review

It’s the end of the lesson, so you should be able to

- Explain what code coverage is, and how different measures differ, including statements, branches, functions, and lines
- Explain the benefits of mutation testing

- When have you written enough tests?
  - When you’ve tested all valid inputs?
  - When your tests get full code coverage?
  - When your tests can catch any buggy mutant code?
  - The answer depends!