CMPT 473
Software Quality Assurance

Mutation Analysis & Testing

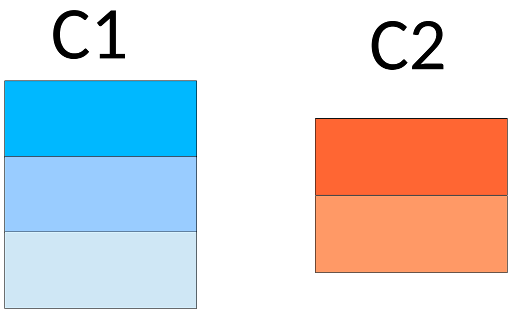
Nick Sumner
With material from Ammann & Offutt, Patrick Lam, Gordon Fraser

How Else Can We Judge Adequacy?

- Input & graph based techniques provide requirements that measure quality.

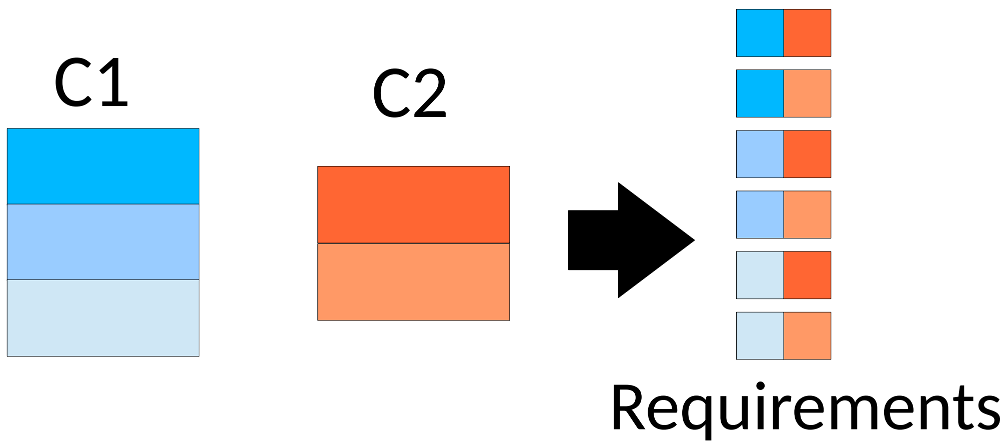
How Else Can We Judge Adequacy?

- *Input* & graph based techniques provide requirements that measure quality.



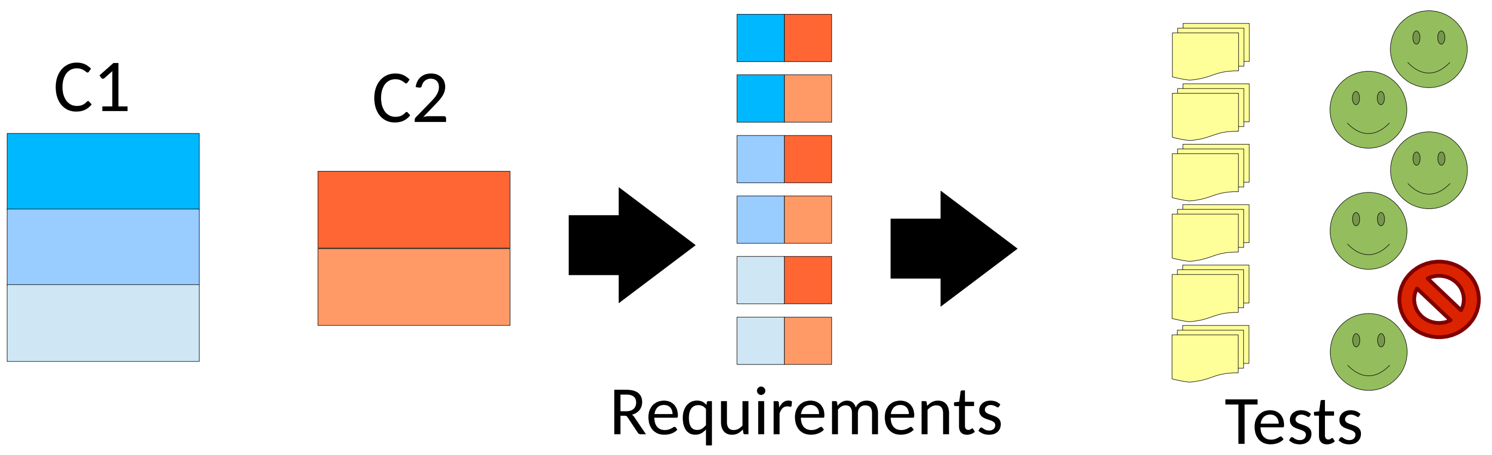
How Else Can We Judge Adequacy?

- *Input* & graph based techniques provide requirements that measure quality.



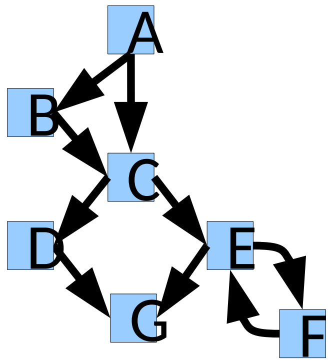
How Else Can We Judge Adequacy?

- *Input* & graph based techniques provide requirements that measure quality.



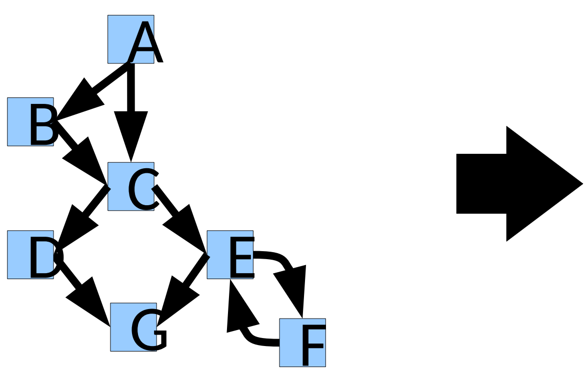
How Else Can We Judge Adequacy?

- Input & *graph* based techniques provide requirements that measure quality.



How Else Can We Judge Adequacy?

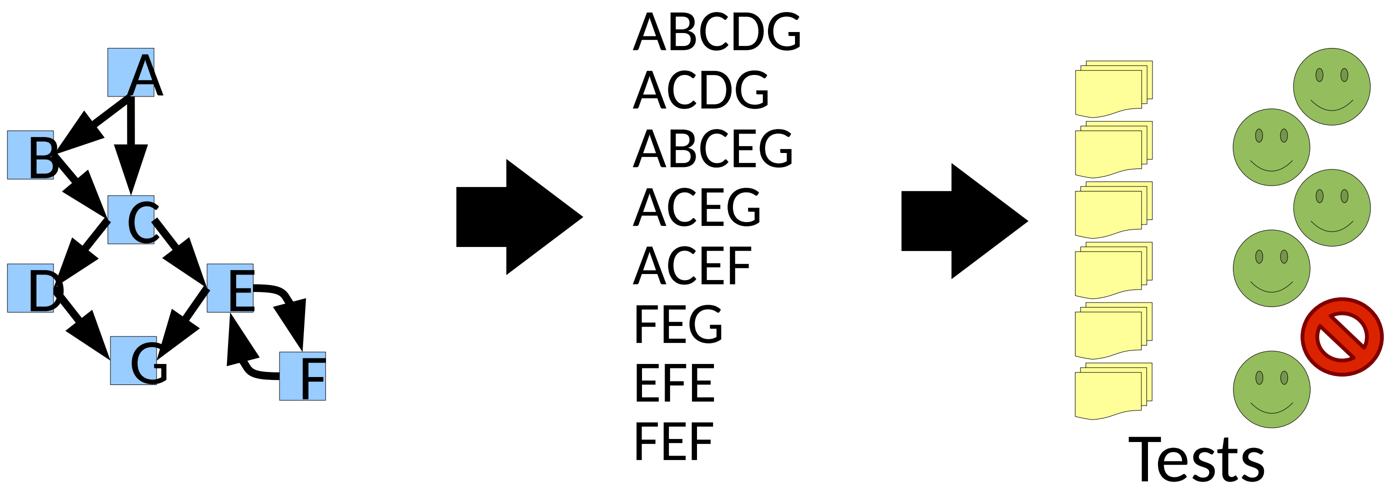
- Input & *graph* based techniques provide requirements that measure quality.



ABCDG
ACDG
ABCEG
ACEG
ACEF
FEG
EFE
FEF

How Else Can We Judge Adequacy?

- Input & *graph* based techniques provide requirements that measure quality.



How Else Can We Judge Adequacy?

- Input & graph based techniques provide requirements that measure quality.
  - But they still have difficulties finding bugs!

How Else Can We Judge Adequacy?

- Input & graph based techniques provide requirements that measure quality.
  - But they still have difficulties finding bugs!
  - Can we try to measure that directly?

How might you go about this?

Fault Seeding

- Insert or ***seed*** representative/typical faults

Fault Seeding

- Insert or ***seed*** representative/typical faults
- Measure how many are found or ***killed*** by the test suite

Fault Seeding

- Insert or ***seed*** representative/typical faults
- Measure how many are found or ***killed*** by the test suite
  - Effectiveness = # killed / # seeded

Fault Seeding

- Insert or ***seed*** representative/typical faults
- Measure how many are found or ***killed*** by the test suite
  - Effectiveness = # killed / # seeded
  - Directly measures *bug finding ability*

Fault Seeding

- Insert or ***seed*** representative/typical faults
- Measure how many are found or ***killed*** by the test suite
  - Effectiveness = # killed / # seeded
  - Directly measures bug finding ability
- Why might this fail?

Fault Seeding

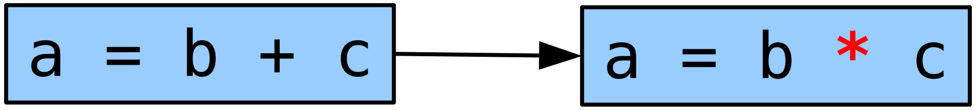
- Insert or ***seed*** representative/typical faults
- Measure how many are found or ***killed*** by the test suite
  - Effectiveness = # killed / # seeded
  - Directly measures bug finding ability
- Why might this fail?
  - What are representative faults?
  - Are there enough faults to be meaningful?
  - Did you forget to remove faults afterward?

Mutation Analysis & Testing

- Mutant
  - A valid program that behaves differently than the original

Mutation Analysis & Testing

- Mutant
  - A valid program that behaves differently than the original
  - Consider small, local changes to programs



Mutation Analysis & Testing

- Mutant
  - A valid program that behaves differently than the original
  - Consider small, local changes to programs
  - A test t kills a mutant m if t produces a different outcome on m than the original program

Mutation Analysis & Testing

- Mutant
  - A valid program that behaves differently than the original
  - Consider small, local changes to programs
  - A test t kills a mutant m if t produces a different outcome on m than the original program

What does this mean?

Mutation Analysis & Testing

- Mutant
  - A valid program that behaves differently than the original
  - Consider small, local changes to programs
  - A test t kills a mutant m if t produces a different outcome on m than the original program
- Systematically generate mutants separately from original program

Mutation Analysis & Testing

- Mutant
  - A valid program that behaves differently than the original
  - Consider small, local changes to programs
  - A test t kills a mutant m if t produces a different outcome on m than the original program
- Systematically generate mutants separately from original program
- The goal is to:
  - **Mutation Analysis** – Measure bug finding ability

Mutation Analysis & Testing

- Mutant
  - A valid program that behaves differently than the original
  - Consider small, local changes to programs
  - A test t kills a mutant m if t produces a different outcome on m than the original program
- Systematically generate mutants separately from original program
- The goal is to:
  - **Mutation Analysis** – Measure bug finding ability
  - **Mutation Testing** – create a test suite that kills a representative set of mutants

Mutation

- What are possible mutants?

```
int foo(int x, int y) {
  if (x > 5) {return x + y;}
  else {return x;}
}
```

Mutation

- What are possible mutants?

```
int foo(int x, int y) {
  if (x > 5) {return x + y;}
  else {return x;}
}
```

- Once we have a test case that kills a mutant, the mutant itself is no longer useful.

Mutation

- What are possible mutants?

```
int foo(int x, int y) {
  if (x > 5) {return x + y;}
  else {return x;}
}
```

- Once we have a test case that kills a mutant, the mutant itself is no longer useful.
- Some are not generally useful:

Why might they not be useful?

Mutation

- What are possible mutants?

```
int foo(int x, int y) {
  if (x > 5) {return x + y;}
  else {return x;}
}
```

- Once we have a test case that kills a mutant, the mutant itself is no longer useful.
- Some are not generally useful:
  - Not compilable

Mutation

- What are possible mutants?

```
int foo(int x, int y) {
  if (x > 5) {return x + y;}
  else {return x;}
}
```

- Once we have a test case that kills a mutant, the mutant itself is no longer useful.
- Some are not generally useful:
  - Not compilable
  - (*Trivial*) Killed by most test cases

Mutation

- What are possible mutants?

```
int foo(int x, int y) {
  if (x > 5) {return x + y;}
  else {return x;}
}
```

- Once we have a test case that kills a mutant, the mutant itself is no longer useful.
- Some are not generally useful:
  - Not compilable
  - (*Trivial*) Killed by most test cases
  - (*Equivalent*) Indistinguishable from original program

Mutation

- What are possible mutants?

```
int foo(int x, int y) {
  if (x > 5) {return x + y;}
  else {return x;}
}
```

- Once we have a test case that kills a mutant, the mutant itself is no longer useful.
- Some are not generally useful:
  - Not compilable
  - (*Trivial*) Killed by most test cases
  - (*Equivalent*) Indistinguishable from original program
  - (*Redundant*) Indistinguishable from other mutants

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;

  if (b < a) {


    minVal = b;




  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
Mutant 1: minVal = b;
  if (b < a) {


    minVal = b;




  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
Mutant 1: minVal = b;
  if (b < a) {
Mutant 2: if (b > a) {
    minVal = b;
  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
Mutant 1: minVal = b;
  if (b < a) {
Mutant 2: if (b > a) {
Mutant 3: if (b < minVal) {
    minVal = b;
  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
Mutant 1: minVal = b;
  if (b < a) {
Mutant 2: if (b > a) {
Mutant 3: if (b < minVal) {
    minVal = b;
Mutant 4:   BOMB();
  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
Mutant 1: minVal = b;
  if (b < a) {
Mutant 2: if (b > a) {
Mutant 3: if (b < minVal) {
    minVal = b;
Mutant 4:   BOMB();
Mutant 5:   minVal = a;

  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
Mutant 1: minVal = b;
  if (b < a) {
Mutant 2: if (b > a) {
Mutant 3: if (b < minVal) {
    minVal = b;
Mutant 4:   BOMB();
Mutant 5:   minVal = a;
Mutant 6:   minVal = failOnZero(b);
  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
Mutant 1: minVal = b;
Mutant 2: if (b > a) {
Mutant 3: if (b < minVal) {
Mutant 4:   BOMB();
ant 5:   minVal = a;
ant 6:   minVal = failOnZero(b);
```

What mimics statement coverage?

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
Mutant 1: minVal = b;
  if (b < a) {
Mutant 2: if (b > a) {
Mutant 3: if (b < minVal) {
    minVal = b;
Mutant 4:   BOMB();
Mutant 5:   minVal = a;
Mutant 6:   minVal = failOnZero(b);
  }
  return minVal;
}
```

What mimics input classes?

- Mimic mistakes
- Encode knowledge from other techniques

Mutation Analysis

Mutants

`Mutant 1`

`Mutant 2`

`Mutant 3`

`Mutant 4`

`Mutant 5`

`Mutant 6`

Mutation Analysis

Mutants

- Mutant 1
- Mutant 2
- Mutant 3
- Mutant 4
- Mutant 5
- Mutant 6

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

Mutation Analysis

Mutants

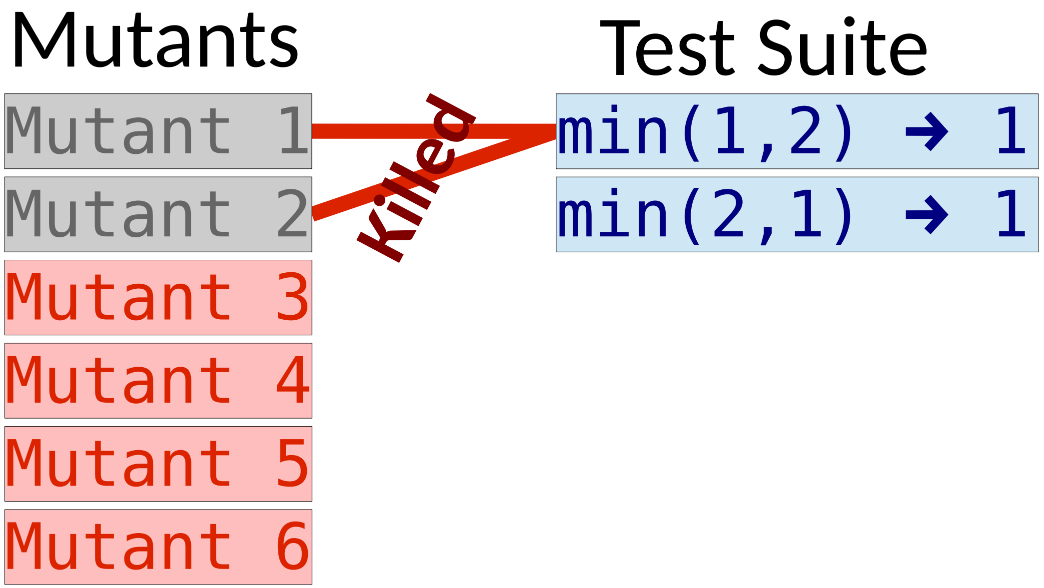
```
Mutant 1
Mutant 2
Mutant 3
Mutant 4
Mutant 5
Mutant 6
```

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

Try every mutant on test 1.

Mutation Analysis



Mutation Analysis

Mutants

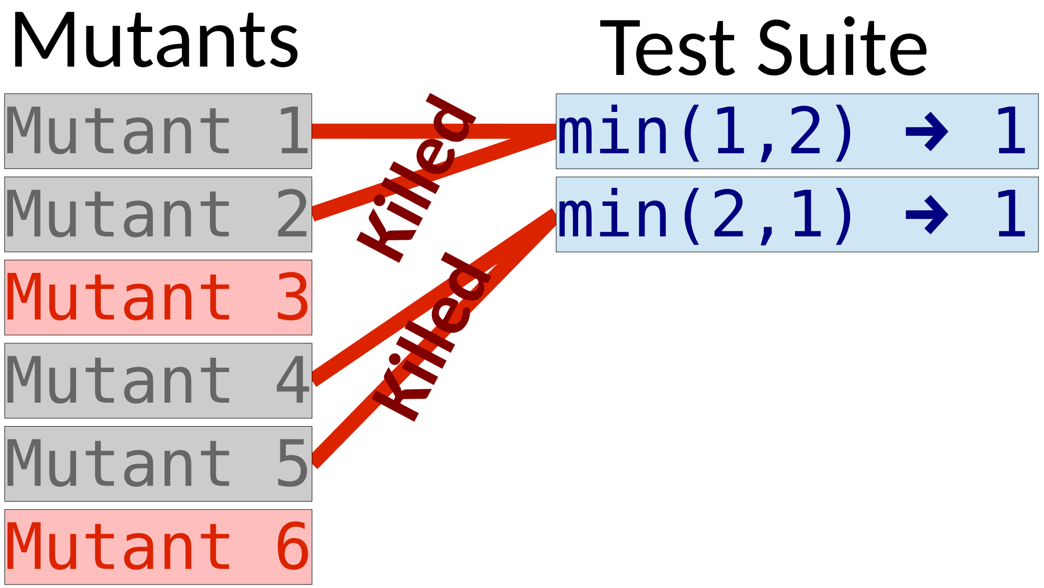
Test Suite

```
Mutant 1    min(1,2) → 1
Mutant 2    min(2,1) → 1
Mutant 3
Mutant 4
Mutant 5
Mutant 6
```

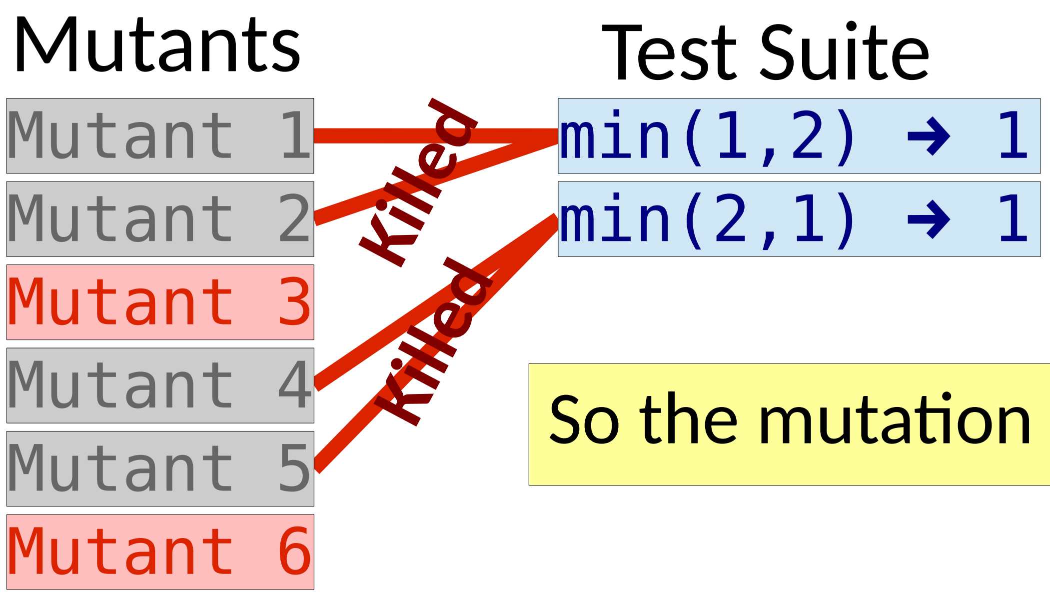
Killed

Try every ***live*** mutant on test 2.

Mutation Analysis

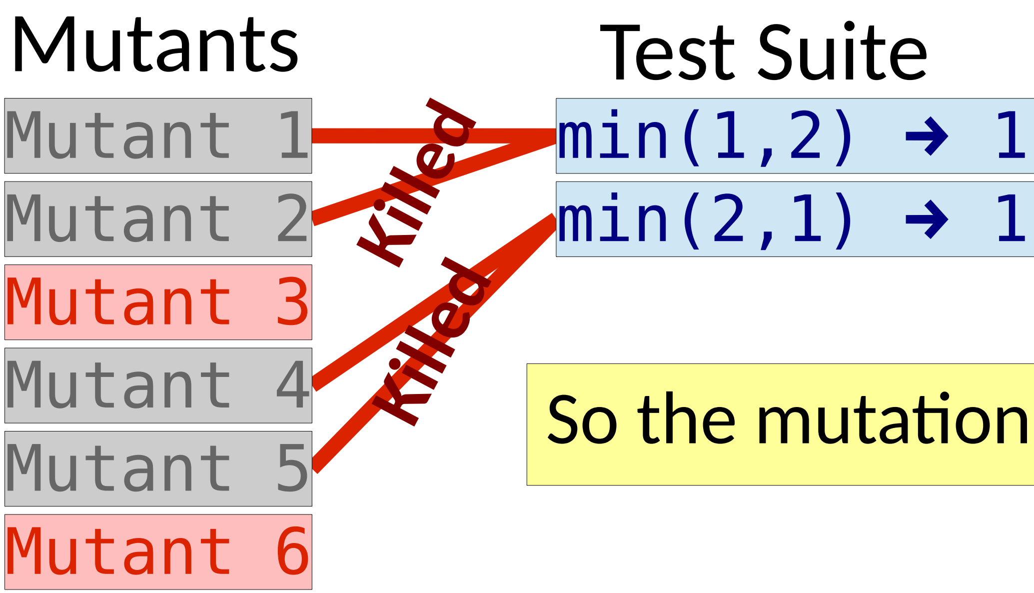


Mutation Analysis



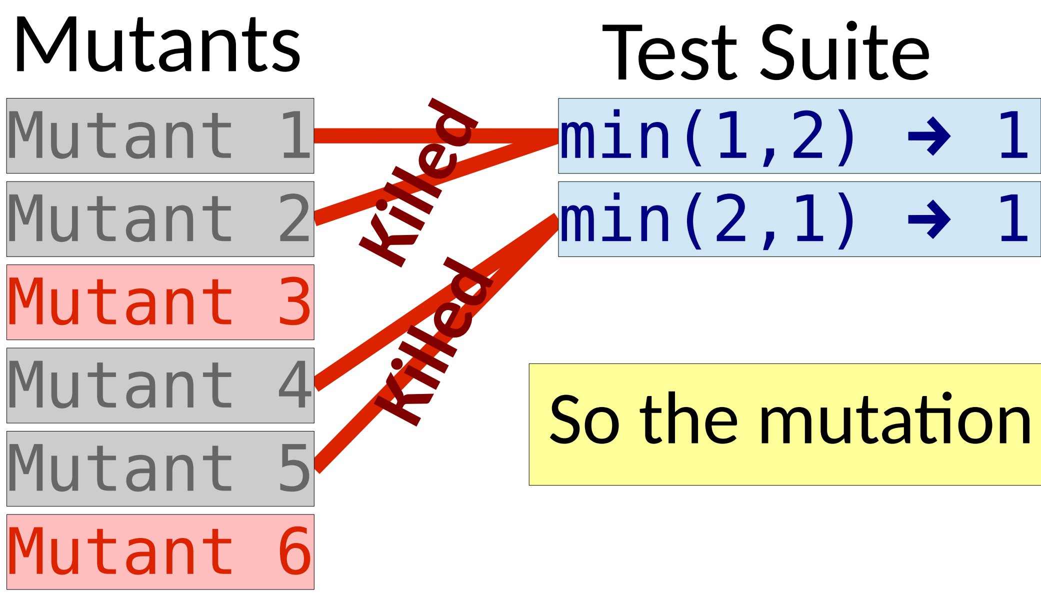
So the mutation score is…

Mutation Analysis



So the mutation score is… **4/5**. *Why?*

Mutation Analysis



So the mutation score is... **4/5**. *Why?*

```
min3(int a, int b):
  int minVal;
  minVal = a;
  if (b < minVal)
    minVal = b;
  return minVal;
```

```
min6(int a, int b):
  int minVal;
  minVal = a;
  if (b < a)
    minVal = failOnZero(b);
  return minVal;
```

Mutation Analysis

Mutants

- Mutant 1
- Mutant 2
- Mutant 3
- Mutant 4
- Mutant 5
- Mutant 6

Test Suite

- `min(1,2) → 1`
- `min(2,1) → 1`

Killed

Killed

So the mutation score is… **4/5**. *Why?*

Equivalent to the original!
There is no injected bug.

```
min3(int a, int b):
  int minVal;
  minVal = a;
  if (b < minVal)
    minVal = b;
  return minVal;
```

```
min3(int a, int b):
  int minVal;
  minVal = a;
  if (b < a)
    minVal = failOnZero(b);
  return minVal;
```

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#Killed}{\#Mutants}$$

Start with the score from fault seeding

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\,\text{Killed}}{\#\,\text{Mutants} - \#\,\text{Equivalent}}$$

Traditional mutation score from literature

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}$$

Updated for modern handling of duplicate & equivalent mutants

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\,\text{Killed} - \#\,\text{Killed Duplicates}}{\#\,\text{Mutants} - \#\,\text{Equivalent} - \#\,\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general
- So why are they equivalent?

Reachability **I**nfection **P**ropagation

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general
- So why are they equivalent?

Reachability **I**nfection **P**ropagation

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general
- So why are they equivalent?

Reachability **I**nfection **P**ropagation

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general
- So why are they equivalent?

Reachability **I**nfection **P**ropagation

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general
- So why are they equivalent?

Reachability **I**nfection ? **P**ropagation

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\,\text{Killed} - \#\,\text{Killed Duplicates}}{\#\,\text{Mutants} - \#\,\text{Equivalent} - \#\,\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general
- So why are they equivalent?

Reachability **I**nfection **P**ropagation

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- New Mutation Score:

$$\frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}$$

- Detecting equivalent mutants is *undecidable* in general
- So why are they equivalent?

More on this later....

Reachability ? **I**nfection ? **P**ropagation

Equivalent Mutants

- Identifying equivalent mutants is one of the most expensive / burdensome aspects of mutation analysis.

Equivalent Mutants

- Identifying equivalent mutants is one of the most expensive / burdensome aspects of mutation analysis.

```
min3(int a, int b):
  int minVal;
  minVal = a;
  if (b < minVal)
    minVal = b;
  return minVal;
```

Requires reasoning about why the result was the same.

Mutation Testing

- Given an unkilled mutant, how can we improve the test suite?

Mutation Testing

- Given an unkilled mutant, how can we improve the test suite?

```
min3(int a, int b):
  int minVal;
  minVal = a;
  if (b < a)
    minVal = failOnZero(b);
  return minVal;
```

Mutation Testing

- Given an unkilled mutant, how can we improve the test suite?

```
min3(int a, int b):
  int minVal;
  minVal = a;
  if (b < a)
    minVal = failOnZero(b);
  return minVal;
```

New Test: `min(2,0) → 0`

New Score: 5/5

Mutation Operators

- The mutants should guide the tester toward an effective test suite

Mutation Operators

- The mutants should guide the tester toward an effective test suite
  - Need a 'representative' pool of mutants

    idea: “If there is a fault, there is a mutant to match it”

Mutation Operators

- The mutants should guide the tester toward an effective test suite
  - Need a 'representative' pool of mutants
    idea: “If there is a fault, there is a mutant to match it”
  - Need a rigorous way of *creating* mutants

Mutation Operators

- The mutants should guide the tester toward an effective test suite
  - Need a 'representative' pool of mutants
    idea: "If there is a fault, there is a mutant to match it"
  - Need a rigorous way of *creating* mutants
- Mutation Operators
  - Systematic changes that may be applied to produce mutants

Mutation Operators

- The mutants should guide the tester toward an effective test suite
  - Need a 'representative' pool of mutants
    idea: "If there is a fault, there is a mutant to match it"
  - Need a rigorous way of *creating* mutants
- Mutation Operators
  - Systematic changes that may be applied to produce mutants
  - Language dependent, but often similar

Mutation Operators

- The mutants should guide the tester toward an effective test suite
  - Need a 'representative' pool of mutants
    idea: “If there is a fault, there is a mutant to match it”
  - Need a rigorous way of *creating* mutants
- Mutation Operators
  - Systematic changes that may be applied to produce mutants
  - Language dependent, but often similar

Why might they be language dependent?

Some Mutation Operators – in Java

- Absolute Value Insertion
  - Each arithmetic (sub)expression is wrapped with `abs()`, `-abs()`, and `failOnZero()`

```
w = x + y + z
```

Just for `abs()`?

Some Mutation Operators – in Java

- Absolute Value Insertion
  - Each arithmetic (sub)expression is wrapped with `abs()`, `-abs()`, and `failOnZero()`

```
w = x + y + z
```

Just for `abs()`?

```
w = abs(x) + y + z
```

```
w = abs(x + y) + z
```

```
w = x + abs(y) + z
```

```
w = x + abs(y + z)
```

```
w = x + y + abs(z)
```

```
w = abs(x + y + z)
```

Just for `abs()`!

Some Mutation Operators – in Java

- Absolute Value Insertion
  - Each arithmetic (sub)expression is wrapped with `abs()`, `-abs()`, and `failOnZero()`
- Arithmetic Operator Replacement
  - Each operator (+,-,*,/,%,...) is replaced with each other operator and LEFTOP and RIGHTOP (returning the named operand).

```
w = x + y + z
```

Some Mutation Operators – in Java

- Absolute Value Insertion
  - Each arithmetic (sub)expression is wrapped with `abs()`, `-abs()`, and `failOnZero()`
- Arithmetic Operator Replacement
  - Each operator (+,-,*,/,%,...) is replaced with each other operator and LEFTOP and RIGHTOP (returning the named operand).

```
w = x + y + z
```

```
w = x + y * z
```

```
w = x + y
```

...

Some Mutation Operators – in Java

- Absolute Value Insertion
  - Each arithmetic (sub)expression is wrapped with `abs()`, `-abs()`, and `failOnZero()`
- Arithmetic Operator Replacement
  - Each operator (+,-,*,/,%,…) is replaced with each other operator and LEFTOP and RIGHTOP (returning the named operand).
- Relational Operator Replacement
  - Each operator (=,!=,<,<=,>,>=) is replaced with each other and TRUEOP and FALSEOP

Some Mutation Operators – in Java

- Conditional Operator Replacement
  - Replace operators (&&, ||, &, |, ^) with each other and LEFTOP, RIGHTOP, TRUEOP, FALSEOP

Some Mutation Operators – in Java

- Conditional Operator Replacement
  - Replace operators (&&, ||, &, |, ^) with each other and LEFTOP, RIGHTOP, TRUEOP, FALSEOP

Could these be used to mimic edge coverage?

Some Mutation Operators – in Java

- Conditional Operator Replacement
  - Replace operators (&&, ||, &, |, ^) with each other and LEFTOP, RIGHTOP, TRUEOP, FALSEOP
- The operator replacement pattern continues…
  - Assignment, Unary Insertion, Unary Deletion

Some Mutation Operators – in Java

- Conditional Operator Replacement
  - Replace operators (&&, ||, &, |, ^) with each other and LEFTOP, RIGHTOP, TRUEOP, FALSEOP
- The operator replacement pattern continues…
  - Assignment, Unary Insertion, Unary Deletion
- Scalar Variable Replacement
  - Replace each variable use with another compatible variable in scope

What does compatible mean? Is it necessary?

Some Mutation Operators – in Java

- Conditional Operator Replacement
  - Replace operators (&&, ||, &, |, ^) with each other and LEFTOP, RIGHTOP, TRUEOP, FALSEOP
- The operator replacement pattern continues…
  - Assignment, Unary Insertion, Unary Deletion
- Scalar Variable Replacement
  - Replace each variable use with another compatible variable in scope
- Bomb Statement Replacement
  - Replace a statement with BOMB()

Some Mutation Operators – in Java

- Conditional Operator Replacement
  - Replace operators (&&, ||, &, |, ^) with each other and LEFTOP, RIGHTOP, TRUEOP, FALSEOP
- The operator replacement pattern continues…
  - Assignme … on

How does the BOMB() operator mimic statement coverage?

- Scalar Variable Replacement
  - Replace each variable use with another compatible variable in scope
- Bomb Statement Replacement
  - Replace a statement with BOMB()

Some Mutation Operators – in Java

- These are all ***intra***procedural (within one method)
- What might ***inter***procedural operators be?

Some Mutation Operators – in Java

- These are all ***intra***procedural (within one method)
- What might ***inter***procedural operators be?
  - Changing parameter values
  - Changing the call target
  - Changing incoming dependencies
  - …

Some Mutation Operators – in Java

- These are all ***intra***procedural (within one method)
- What might ***inter***procedural operators be?
  - Changing parameter values
  - Changing the call target
  - Changing incoming dependencies
  - …
- And more…
  - Interface Mutation, Object Oriented Mutation, …

Some Mutation Operators – in Java

- These are all ***intra***procedural (within one method)
- What might ***inter***procedural operators be?
  - Changing parameter values
  - Changing the call target
  - Changing incoming dependencies
  - …
- And more…
- Often just the simplest are used

Mutation Operators

- Are the mutants representative of all bugs?
- Do we expect the mutation score to be meaningful?

Ideas? Why? Why not?

Mutation Operators

- Are the mutants representative of all bugs?
- Do we expect the mutation score to be meaningful?

Ideas? Why? Why not?

2 Key ideas are missing….

Competent Programmer Hypothesis

Programmers *tend* to write code that is *almost* correct

Competent Programmer Hypothesis

Programmers *tend* to write code that is *almost* correct

- So *most* of the time simple mutations should reflect the real bugs.

Coupling Effect

Tests that cover so much behavior that even simple errors are detected should also be sensitive enough to detect more complex errors

Coupling Effect

Tests that cover so much behavior that even simple errors are detected should also be sensitive enough to detect more complex errors

- By casting a fine enough net, we'll catch the big fish, too
  (sorry dolphins)

What Problems Remain?

- Scale (there are a lot of tests)

What Problems Remain?

- Scale (there are a lot of tests)
- Equivalence

What Problems Remain?

- Scale (there are a lot of tests)
- Equivalence

- Scale may be attacked in many ways

Ideas?

What Problems Remain?

- Scale (there are a lot of tests)
- Equivalence

- Scale may be attacked in many ways
  - Coverage filters
  - Short circuiting tests
  - Testing mutants simultaneously

What Problems Remain?

- Scale (there are a lot of tests)
- Equivalence

- Scale may be attacked in many ways
  - Coverage filters
  - Short circuiting tests
  - Testing mutants simultaneously
- Can also modify *mutation criteria* to help with *both…*

Mutation Criteria

- Recall: If a test can detect a mutant, that mutant is ***killed*** by the test.

Mutation Criteria

- Recall: If a test can detect a mutant, that mutant is ***killed*** by the test.

What does it mean if a mutant was killed?

Mutation Criteria

- Recall: If a test can detect a mutant, that mutant is ***killed*** by the test.

What does it mean if a mutant was killed?

What does it mean if a mutant was ***not*** killed?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different ***output*** than p(t)

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different ***output*** than p(t)

Reachability **I**nfection **P**ropagation

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different ***output*** than p(t)
- Weakly Killed
  - A test weakly kills a mutant m if m(t) produces different ***internal state*** than p(t)

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different ***output*** than p(t)
- Weakly Killed
  - A test weakly kills a mutant m if m(t) produces different ***internal state*** than p(t)

Reachability **I**nfection **P**ropagation ?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different ***output*** than p(t)
- Weakly Killed
  - A test weakly kills a mutant m if m(t) produces different ***internal state*** than p(t)
  - Reachable, infects, but might not propagate.

How might this happen?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

n if m(t) produces different

t not propagate.

How might this happen?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

`a = 10, b = 5`

n if m(t) produces different

n't propagate.

How might this happen?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

`a = 10, b = 5`

`minVal = 5`

n if m(t) produces different

n't propagate.

How might this happen?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

`a = 10, b = 5`

`minVal = 5`

n if m(t) produces different

`minVal = 5`

n't propagate.

How might this happen?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

a = 10, b = 5

minVal = 5

n if m(t) produces different

minVal = 5

n't propagate.

return 5

How might this happen?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

n if m(t) produces different

n't propagate.

How can we strongly kill the mutant instead?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

`a = 5, b = 10`

n if m(t) produces different

n't propagate.

How can we strongly kill the mutant instead?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

`a = 5, b = 10`

`minVal = 10`

n if m(t) produces different

n't propagate.

How can we strongly kill the mutant instead?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = b; // was a
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

`a = 5, b = 10`

`minVal = 10`

n if m(t) produces different

n't propagate.

`return 10`

How can we strongly kill the mutant instead?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

ant m if m(t) produces different

night not propagate.

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

What might an equivalent mutant look like?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < minVal) {
    minVal = b;
  }
  return minVal;
}
```

What might an equivalent mutant look like?

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < a) {
    minVal = b;
  }
  return minVal;
}
```

```
int min(int a, int b) {
  int minVal;
  minVal = a;
  if (b < minVal) {
    minVal = b;
  }
  return minVal;
}
```

They always behave the same way!

Mutation Criteria

- Strongly Killed
  - A test *strongly* kills a mutant m if m(t) produces different ***output*** than p(t)
- Weakly Killed
  - A test weakly kills a mutant m if m(t) produces different ***internal state*** than p(t)
  - Reachable, infects, but might not propagate.

Leading to…

Mutation Criteria

- Strong Mutation Coverage
  - For each mutant, the test suite contains a test that strongly kills the mutant

Mutation Criteria

- Strong Mutation Coverage
  - For each mutant, the test suite contains a test that strongly kills the mutant
- Weak Mutation Coverage
  - For each mutant, the test suite contains a test that weakly kills the mutant

Mutation Criteria

- Strong Mutation Coverage
  - For each mutant, the test suite contains a test that strongly kills the mutant
- Weak Mutation Coverage
  - For each mutant, the test suite contains a test that weakly kills the mutant

How might weak coverage help with equivalence?

Mutation Criteria

- Strong Mutation Coverage
  - For each mutant, the test suite contains a test that strongly kills the mutant
- Weak Mutation Coverage
  - For each mutant, the test suite contains a test that weakly kills the mutant

How might weak coverage help with equivalence?

How might weak coverage help with scalability?

Mutation Criteria

- Strong Mutation Coverage
  - For each mutant, the test suite contains a test that strongly kills the mutant
- Weak Mutation Coverage
  - For each mutant, the test suite contains a test that weakly kills the mutant

How might weak coverage help with equivalence?

How might weak coverage help with scalability?

Is there any reason to prefer strong coverage?

Mutation Testing

- Considered one of the strongest criteria

Why?

Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some graph coverage (node, edge, …)

Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some graph coverage (node, edge, …)
- Massive number of criteria.

Why?

Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some graph coverage (node, edge, …)
- Massive number of criteria.
- Still not always the most tests.

Why?

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$?

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$ T1 is *more likely* to find more bugs.

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$ T1 is *more likely* to find more bugs.

What if you change |T|?

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$ T1 is *more likely* to find more bugs.
  - $Cov_{stmt}(T)$ increases with the $|T|$

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$ T1 is *more likely* to find more bugs.
  - $Cov_{stmt}(T)$ increases with the $|T|$
  - Shrinking $|T|$ while preserving $Cov_{stmt}(T)$ decreases defect finding power

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$ T1 is *more likely* to find more bugs.
  - $Cov_{stmt}(T)$ increases with the $|T|$
  - Shrinking $|T|$ while preserving $Cov_{stmt}(T)$ decreases defect finding power
    → You cannot assume that better coverage increases defect finding ability!

Then does coverage serve a purpose?

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$ T1 is *more likely* to find more bugs.
  - $Cov_{stmt}(T)$ increases with the $|T|$
  - Shrinking $|T|$ while preserving $Cov_{stmt}(T)$ decreases defect finding power
  - Coverage still tells you which portions of a program haven't been tested!

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $Cov_{stmt}(T1) > Cov_{stmt}(T2) \rightarrow$ T1 is *more likely* to find more bugs.
  - $Cov_{stmt}(T)$ increases with the $|T|$
  - Shrinking $|T|$ while preserving $Cov_{stmt}(T)$ decreases defect finding power
  - Coverage still tells you which portions of a program haven't been tested!
  - It just cannot fully measure defect finding capability.

Traditional Coverage vs Mutation

- Mutation analysis/testing correlates with defect finding independent of code coverage! [Just 2014]

Traditional Coverage vs Mutation

- Mutation analysis/testing correlates with defect finding independent of code coverage! [Just 2014]

So is that it?
Can we just do mutation testing & be done?

A Summary of Test Adequacy

- ***Many*** ways of measuring test adequacy.

A Summary of Test Adequacy

- ***Many*** ways of measuring test adequacy.
- ***No*** single ***approach*** is ***sufficient***.

A Summary of Test Adequacy

- ***Many*** ways of measuring test adequacy.
- ***No*** single ***approach*** is ***sufficient***.
- ***Mutation testing*** is the strongest known single approach we presently have, but it comes at a price.

A Summary of Test Adequacy

- ***Many*** ways of measuring test adequacy.
- ***No*** single ***approach*** is ***sufficient***.
- ***Mutation testing*** is the strongest known single approach we presently have, but it comes at a price.
- Even combining all adequacy measures, there will still be bugs.

A Summary of Test Adequacy

- ***Many*** ways of measuring test adequacy.
- ***No*** single ***approach*** is ***sufficient***.
- ***Mutation testing*** is the strongest known single approach we presently have, but it comes at a price.
- Even combining all adequacy measures, there will still be bugs.
  - And they have consequences http://arstechnica.com/security/2016/02/extremely-severe-bug-leaves-dizzying-number-of-apps-and-devices-vulnerable/