CMPT 745
Software Engineering

An Overview of Software Testing

Nick Sumner
wsumner@sfu.ca
Software Testing

- The most common way of measuring and ensuring program correctness
Software Testing

- The most common way of measuring and ensuring program correctness
Software Testing

- The most common way of measuring and ensuring program correctness
Software Testing

- The most common way of measuring and ensuring program correctness
Software Testing

- The most common way of measuring and ensuring program correctness
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues

Test Suite

- Test 1: Input, Oracle
- Test 2: Input, Oracle
- Test 3: Input, Oracle
- Test 4: Input, Oracle
- Test 5: Input, Oracle
- Test 6: Input, Oracle
- Test 7: Input, Oracle
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy

Test Suite
- Test 1: Input → Oracle
- Test 2: Input → Oracle
- Test 3: Input → Oracle
- Test 4: Input → Oracle
- Test 5: Input → Oracle
- Test 6: Input → Oracle
- Test 7: Input → Oracle
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy

Testing is sampling.
How do we know whether we are sampling well?
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy
- Automated input generation

Test Suite
- Test 1: Input, Oracle
- Test 2: Input, Oracle
- Test 3: Input, Oracle
- Test 4: Input, Oracle
- Test 5: Input, Oracle
- Test 6: Input, Oracle
- Test 7: Input, Oracle
Software Testing

- The most common way of measuring and ensuring program correctness

Input → Program → Observed Behavior → Oracle → Outcome

Key Issues
- Test suite adequacy
- Automated input generation
- Automated oracle generation

Test Suite
- Test 1: Input → Oracle
- Test 2: Input → Oracle
- Test 3: Input → Oracle
- Test 4: Input → Oracle
- Test 5: Input → Oracle
- Test 6: Input → Oracle
- Test 7: Input → Oracle
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
- Regression test selection

Test Suite

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Input</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 2</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 3</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 4</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 5</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 6</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 7</td>
<td>Input</td>
<td>Oracle</td>
</tr>
</tbody>
</table>
Software Testing

- The most common way of measuring and ensuring program correctness

Input → Program → Observed Behavior → Oracle → Outcome

Key Issues
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
- Regression test selection
- Fault localization & automated debugging

Test Suite
- Test 1: Input Oracle
- Test 2: Input Oracle
- Test 3: Input Oracle
- Test 4: Input Oracle
- Test 5: Input Oracle
- Test 6: Input Oracle
- Test 7: Input Oracle
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
- Regression test selection
- Fault localization & automated debugging
- Automated program repair
- ...

Test Suite
- Test 1: Input, Oracle
- Test 2: Input, Oracle
- Test 3: Input, Oracle
- Test 4: Input, Oracle
- Test 5: Input, Oracle
- Test 6: Input, Oracle
- Test 7: Input, Oracle
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
- Regression test selection
- Fault localization & automated debugging
- Automated program repair
- ...

We will discuss a few basics now and revisit the problem as we learn new techniques.
Test Suite Design

- **Objectives**
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- **Objectives**
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- **Components – The Automated Testing Pyramid**
Designing a Unit Test

- Common structure
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
  Environment env;
  ExprTree tree;
  auto result = evaluate(tree, env);
  CHECK(!result.has_value());
}
```

Set up a scenario
Designing a Unit Test

- **Common structure**

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);

    CHECK(!result.has_value());
}
```

Run the scenario
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```

Check the outcome
Designing a Unit Test

- Common structure

- Tests should run in isolation

```c
struct Frob {
    Frob()
        : conn{getDB().connect()}
    {
    }
    DBConnection conn;
};
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
  Frob()
  : conn{getDB().connect()}
  {}
  DBConnection conn;
};

TEST_CASE("bad test 1") {
  Frob frob;
  ...
}

TEST_CASE("bad test 2") {
  Frob frob;
  ...
}
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```c
struct Frob {
    Frob()
        : conn{getDB().connect()}
        {}
    DBConnection conn;
};

TEST_CASE("bad test 1") {
    Frob frob;
    ...
}

TEST_CASE("bad test 2") {
    Frob frob;
    ...
}
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```c
struct Frob {
    Frob() : conn{getDB().connect()}
    {
    } 
    DBConnection conn;
};

TEST_CASE("bad test 1") {
    Frob frob;
    ...
}

TEST_CASE("bad test 2") {
    Frob frob;
    ...
}
```

The order of the test can affect the results!
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob()
        : conn{getDB().connect()}
        {}
    DBConnection conn;
};
```

```cpp
test_case("bad test 1") {
    Frob frob;
    ...
}

test_case("bad test 2") {
    Frob frob;
    ...
}
```

The order of the test can affect the results!

A flaky DB can affect results!
Designing a Unit Test

- Common structure
- Tests should run in isolation!
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn) : conn{inConn}
    {
    }
    Connection& conn;
};
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
  Frob(Connection& inConn) :
    conn{inConn}
  {}
  Connection& conn;
};
```

Dependency injection allows the user of a class to control its behavior.
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn) :
        conn{inConn}
    {}
    Connection& conn;
};
```

*Dependency injection* allows the user of a class to control its behavior.
Designing a Unit Test

- **Common structure**

- **Tests should run in isolation**

```cpp
struct Frob {
    Frob(Connection& inConn)
    : conn{inConn}
    {}

    Connection& conn;
};
```

*Dependency injection* allows the user of a class to control its behavior.
Designing a Unit Test

- Common structure

- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn) : conn{inConn} {}
    Connection& conn;
};
```

Dependency injection allows the user of a class to control its behavior.
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
  Frob(Connection& inConn) :
    conn{inConn}
  {}
  Connection& conn;
};

TEST_CASE("better test 1") {
  FakeDB db;
  FakeConnection conn = db.connect();
  Frob frob{conn};
  ...
}
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn) : conn{inConn} {}
    Connection& conn;
};
```

```cpp
test_case("better test 1") {
    FakeDB db;
    FakeConnection conn = db.connect();
    Frob frob{conn};
    ...
}
```

Mocks & stubs isolate and examine how a component interacts with dependencies.
Designing a Unit Test

- Common structure
- Tests should run in isolation
- Key problem to resolve:
  - How do you define your inputs & oracles?
Selecting Inputs

- Two broad categories
Selecting Inputs

- Two broad categories
  - *Black box testing* – treat the program as opaque/unknown
Selecting Inputs

- **Two broad categories**
  - *Black box testing* – treat the program as opaque/unknown
    - specification based (BDD?)
    - model driven
    - naive fuzzing
    - boundary value analysis
Selecting Inputs

- **Two broad categories**
  - *Black box testing* – treat the program as opaque/unknown
  - *White box testing* – program structure & semantics can be used
Selecting Inputs

- Two broad categories
  - *Black box testing* – treat the program as opaque/unknown
  - *White box testing* – program structure & semantics can be used

  symbolic execution
  call chain synthesis
  grey/whitebox fuzzing
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

What about tasks like: machine learning simulation ...

Diagram: 
- Input
- Program
- Observed Behavior
- Oracle
- Outcome
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- Invariants & properties are powerful
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- Invariants & properties are powerful
  - $\text{foo}^{-1}(\text{foo}(x)) = x$ (e.g. archive & unarchive a file)
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- **Invariants & properties are powerful**
  - \( \text{foo}^{-1}(\text{foo}(x)) = x \) (e.g. archive & unarchive a file)
  - \( \text{turn}(360, \text{direction}) = \text{direction} \)
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- Invariants & properties are powerful
  - $\text{foo}^{-1}(\text{foo}(x)) = x$ (e.g. archive & unarchive a file)
  - $\text{turn}(360, \text{direction}) = \text{direction}$

Metamorphic testing
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- Invariants & properties are powerful
  - $\text{foo}^{-1}(\text{foo}(x)) = x$ (e.g. archive & unarchive a file)
  - $\text{turn}(360, \text{direction}) = \text{direction}$
  - $\text{program1}(x) = \text{program2}(x)$
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- Invariants & properties are powerful
  - $\text{foo}^{-1}(\text{foo}(x)) = x$ (e.g. archive & unarchive a file)
  - $\text{turn}(360, \text{direction}) = \text{direction}$
  - $\text{program1}(x) = \text{program2}(x)$

Differential testing
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- Invariants & properties are powerful
  - $\text{foo}^{-1}(\text{foo}(x)) = x$ (e.g. archive & unarchive a file)
  - $\text{turn}(360, \text{direction}) = \text{direction}$
  - $\text{program1}(x) = \text{program2}(x)$

General invariants can be exploited in (semi)automated test generation (e.g. property based)
Designing Oracles

- Sometimes it is simple
  - For a known scenario, a specific output is expected

- Invariants & properties are powerful
  - foo\(^{-1}\)(foo(x)) == x (e.g. archive & unarchive a file)
  - turn(360, direction) == direction
  - program1(x) == program2(x)

- Fully automated tests benefit from fully automated oracles
  - But the problem is hard
Test Suite Adequacy

- A test suite should provide a metric on software quality
Test Suite Adequacy

- A test suite should provide a metric on software quality
  - Passing a test should increase the metric
  - Failing a test should decrease the metric
Test Suite Adequacy

- A test suite should provide a metric on software quality
  - Passing a test should increase the metric
  - Failing a test should decrease the metric

- But a test suite *samples* from the input space
Test Suite Adequacy

- A test suite should provide a metric on software quality
  - Passing a test should increase the metric
  - Failing a test should decrease the metric

- But a test suite *samples* from the input space
  - Is it representative/biased?
Test Suite Adequacy

- A test suite should provide a metric on software quality
  - Passing a test should increase the metric
  - Failing a test should decrease the metric

- But a test suite *samples* from the input space
  - Is it representative/biased?
  - Can we know?
Test Suite Adequacy

- A test suite should provide a metric on software quality
  - Passing a test should increase the metric
  - Failing a test should decrease the metric

- But a test suite *samples* from the input space
  - Is it representative/biased?
  - Can we know?
  - Can we measure how likely a test suite is to measure what we want?
Test Suite Adequacy

- A test suite should provide a metric on software quality
  - Passing a test should increase the metric
  - Failing a test should decrease the metric

- But a test suite *samples* from the input space
  - Is it representative/biased?
  - Can we know?
  - Can we measure how likely a test suite is to measure what we want?

- High level decision making
  - Is a test suite good enough? (Will a higher score mean fewer defects?)
Test Suite Adequacy

- A test suite should provide a metric on software quality
  - Passing a test should increase the metric
  - Failing a test should decrease the metric

- But a test suite *samples* from the input space
  - Is it representative/biased?
  - Can we know?
  - Can we measure how likely a test suite is to measure what we want?

- **High level decision making**
  - Is a test suite good enough? (Will a higher score mean fewer defects?)
  - What parts of a program should be tested better?
Test Suite Adequacy

- Metrics

  **Remember:** A higher score *should* mean fewer defects
Test Suite Adequacy

- **Metrics**
  - Statement coverage

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

Is each *statement covered* by at least one test in the test suite?

\[
\text{score} = \frac{\text{# covered}}{\text{# statements}}
\]
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

We will discuss *control flow graphs* again soon.


def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')

We will discuss *control flow graphs* again soon.

\[
score = \frac{\text{# covered}}{\text{# branches}}\]
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
      ...
    else:
      ...
    print('awesome')
```

Is each branch covered by at least one test in the test suite?

\[
\text{score} = \frac{\# \text{ covered}}{\# \text{ branches}}
\]
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage

Is each *branch covered* by at least one test in the test suite?

score = \( \frac{\text{# covered}}{\text{# branches}} \)

```python
def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage

It is widely agreed that statement/edge coverage are not good *measures*.

But they are *sanity checks*.

Test suite adequacy is complex. [Groce 2014]

\[
\text{score} = \frac{\text{# covered}}{\text{# branches}}
\]
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - **MC/DC coverage***

Does each *term determine* the outcome of at least one condition in the test suite?

```python
def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - **MC/DC coverage**

Does each *term determine* the outcome of at least one condition in the test suite?

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*

Does each *term determine* the outcome of at least one condition in the test suite?

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

- a=#T  b=#T  c=#F  ↦  #T
- a=#F  b=#T  c=#F  ↦  #F
Test Suite Adequacy

- Metrics
  - Statement coverage
  - Branch coverage
  - **MC/DC coverage**

Does each term determine the outcome of at least one condition in the test suite?

```python
def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

- $a=\#T$, $b=\#T$, $c=\#F$ $\mapsto$ $\#T$
- $a=\#F$, $b=\#T$, $c=\#F$ $\mapsto$ $\#F$

**a** in this condition is covered by the test suite
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - **MC/DC coverage**

Does each *term determine* the outcome of at least one condition in the test suite?

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

Required by regulation in (e.g.) avionics, safety critical systems, automotive software.
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - **Mutation coverage***

```python
def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

How many *injected bugs* can be detected by the test suite?
Test Suite Adequacy

- Metrics
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - Mutation coverage*

def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')

How many injected bugs can be detected by the test suite?
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - Mutation coverage*

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

How many **injected bugs** can be detected by the test suite?
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - Mutation coverage*

How many *injected bugs* can be detected by the test suite?

\[
\text{score} = \frac{\# \text{ covered/killed}}{\# \text{ non-equivalent mutants}}
\]
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - Mutation coverage*
  - Path coverage
  - ...

Is each *path covered* by at least one test in the test suite?

```python
def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - Mutation coverage*
  - Path coverage
  - ...

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

Is each path covered by at least one test in the test suite?
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - Mutation coverage*
  - Path coverage
  - ...

Is each *path covered* by at least one test in the test suite?

```python
def my_lovely_fun(a, b, c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

![Diagram of paths and test cases]
Test Suite Adequacy

- **Metrics**
  - Statement coverage
  - Branch coverage
  - MC/DC coverage*
  - Mutation coverage*
  - Path coverage
  - ...

But shrinking test suites while maintaining St, Br, MC/DC decreases defect detection.

There is more going on here.

[Rothermel 1998, Yoo 2012, Shi 2018]
MC/DC Testing
MC/DC Coverage

- Logic & conditional behaviors are pervasive
MC/DC Coverage

- Logic & conditional behaviors are pervasive
- *if* statements are the most frequently fixed statements in bug fixes
  [Pan, ESE 2008]
MC/DC Coverage

- Logic & conditional behaviors are pervasive
- if statements are the most frequently fixed statements in bug fixes [Pan, ESE 2008]
- **Safety critical systems often involve many complex conditions** (avionics, medical, automotive, ...)

MC/DC Coverage

- Logic & conditional behaviors are pervasive
- `if` statements are the most frequently fixed statements in bug fixes [Pan, ESE 2008]
- Safety critical systems often involve many complex conditions (avionics, medical, automotive, ...)
- We should place more effort/burden on ensuring correctness of conditions
MC/DC Coverage

- A *predicate* is simply a boolean expression.
MC/DC Coverage

- A *predicate* is simply a boolean expression.

- *Predicate Coverage* requires each predicate to be true in one test & be false in one test.
MC/DC Coverage

- A *predicate* is simply a boolean expression.
- *Predicate Coverage* requires each predicate to be true in one test & be false in one test.

```
if (a || b) && (c || d):
    s
```

```
if (a | b) & (c | d):
    s
```

How does it do in these cases?
MC/DC Coverage

- A *predicate* is simply a boolean expression.
- *Predicate Coverage* requires each predicate to be true in one test & be false in one test.

if (a || b) && (c || d):
  s

if (a | b) & (c | d):
  s
A *predicate* is simply a boolean expression.

*Predicate Coverage* requires each predicate to be true in one test & be false in one test.

*Clause Coverage* requires each clause to be true in one test & be false in one test.
MC/DC Coverage

- A **predicate** is simply a boolean expression.

- **Predicate Coverage** requires each predicate to be true in one test & be false in one test.

- **Clause Coverage** requires each clause to be true in one test & be false in one test.

```
if (a || b) && (c || d):
s
if (a | b) & (c | d):
s
```

How does it do in these cases?
MC/DC Coverage

- A **predicate** is simply a boolean expression.
- **Predicate Coverage** requires each predicate to be true in one test & be false in one test.
- **Clause Coverage** requires each clause to be true in one test & be false in one test.
MC/DC Coverage

- A **predicate** is simply a boolean expression.
- **Predicate Coverage** requires each predicate to be true in one test & be false in one test.
- **Clause Coverage** requires each clause to be true in one test & be false in one test.

```
if (a || b) && (c || d):
```

![Table of truth values for if (a || b) && (c || d)]

```
if (a | b) & (c | d):
```

![Table of truth values for if (a | b) & (c | d)]

How many tests?
MC/DC Coverage

- A **predicate** is simply a boolean expression
- **Predicate Coverage** requires each predicate to be true in one test & be false in one test.
- **Clause Coverage** requires each clause to be true in one test & be false in one test.

\[
\text{if } (a \lor b) \land (c \lor d):
\]

Minimum of 2 tests
MC/DC Coverage

- A **predicate** is simply a boolean expression
- **Predicate Coverage** requires each predicate to be true in one test & be false in one test.
- **Clause Coverage** requires each clause to be true in one test & be false in one test.

```
if (a || b) && (c || d):
  s
```

<table>
<thead>
<tr>
<th>T</th>
<th>F</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Minimum of 2 tests

- a=true, b=true, c=false, d=false
- a=false, b=false, c=true, d=true
A predicate is simply a boolean expression.

Predicate Coverage requires each predicate to be true in one test & be false in one test.

Clause Coverage requires each clause to be true in one test & be false in one test.

Minimum of 2 tests:
- a=true, b=true, c=false, d=false
- a=false, b=false, c=true, d=true
MC/DC Coverage

- Modified Condition/Decision Coverage
MC/DC Coverage

- Modified Condition/Decision Coverage
  1) Each entry & exit is used
MC/DC Coverage

- Modified Condition/Decision Coverage
  1) Each entry & exit is used
  2) Each decision/branch takes every possible outcome
MC/DC Coverage

- **Modified Condition/Decision Coverage**
  1) Each entry & exit is used
  2) Each decision/branch takes every possible outcome
  3) Each clause takes every possible outcome
MC/DC Coverage

- **Modified Condition/Decision Coverage**
  1) Each entry & exit is used
  2) Each decision/branch takes every possible outcome
  3) Each clause takes every possible outcome

So far, this is clause coverage w/o that pathological case
**MC/DC Coverage**

- **Modified Condition/Decision Coverage**
  1. Each entry & exit is used
  2. Each decision/branch takes every possible outcome
  3. Each clause takes every possible outcome

```java
if (a || b) && (c || d):
```

Minimum of 2 tests

a=\text{true}, b=\text{true}, c=\text{true}, d=\text{true}

a=\text{false}, b=\text{false}, c=\text{false}, d=\text{false}
**MC/DC Coverage**

- **Modified Condition/Decision Coverage**
  1. Each entry & exit is used
  2. Each decision/branch takes every possible outcome
  3. Each clause takes every possible outcome

If \( a \lor b \) \&\& \( c \lor d \):

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is this good?

Minimum of 2 tests

- \( a = \text{true}, b = \text{true}, c = \text{true}, d = \text{true} \)
- \( a = \text{false}, b = \text{false}, c = \text{false}, d = \text{false} \)
Modified Condition/Decision Coverage

1) Each entry & exit is used
2) Each decision/branch takes every possible outcome
3) Each clause takes every possible outcome
4) Each clause independently impacts the outcome
MC/DC Coverage

- **Modified Condition/Decision Coverage**
  1. Each entry & exit is used
  2. Each decision/branch takes every possible outcome
  3. Each clause takes every possible outcome
  4. Each clause independently impacts the outcome

**Intuition:**
Make sure that the tests for one clause are not *hidden* by other clauses.
MC/DC Coverage

- Modified Condition/Decision Coverage
  1) Each entry & exit is used
  2) Each decision/branch takes every possible outcome
  3) Each clause takes every possible outcome
  4) Each clause independently impacts the outcome

- Use in safety critical systems: avionics, spacecraft, ...
MC/DC Coverage

- **Modified Condition/Decision Coverage**
  1) Each entry & exit is used
  2) Each decision/branch takes every possible outcome
  3) Each clause takes every possible outcome
  4) Each clause independently impacts the the outcome

- Use in safety critical systems: avionics, spacecraft, ...

- Not only ensures that clauses are tested, but that each *has an impact*
MC/DC Coverage

- A clause determines the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate
A clause determines the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate.

```python
def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
    print('awesome')
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>#T</td>
<td>#T</td>
<td>#F</td>
<td>#T</td>
</tr>
<tr>
<td>#F</td>
<td>#T</td>
<td>#F</td>
<td>#F</td>
</tr>
</tbody>
</table>
MC/DC Coverage

- A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate

\[ \varphi(a,b,c) \neq \varphi(a,b,\neg c) \]
MC/DC Coverage

- A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate

\[ \varphi(a,b,c) \neq \varphi(a,b,\neg c) \]

\[(a \mid\mid b \&\& c)\]
MC/DC Coverage

- A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate.

\[ \varphi(a, b, c) \neq \varphi(a, b, \neg c) \]
\[ (a \mid\mid b \&\& c) \]

\[
\begin{align*}
a &= F \\
b &= T \\
c &= T \\
T
\end{align*}
\]
MC/DC Coverage

- A clause **determines** the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate

\[ \varphi(a,b,c) \neq \varphi(a,b,-c) \]

\[(a \mid \mid b \&\& c)\]

<table>
<thead>
<tr>
<th>a=F</th>
<th>a=F</th>
</tr>
</thead>
<tbody>
<tr>
<td>b=T</td>
<td>b=T</td>
</tr>
<tr>
<td>c=T</td>
<td>c=F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
A clause **determines** the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate

\[ \varphi(a,b,c) \neq \varphi(a,b,\neg c) \]

\[(a \mid\mid b \&\& c)\]

<table>
<thead>
<tr>
<th>a=F</th>
<th>a=F</th>
</tr>
</thead>
<tbody>
<tr>
<td>b=T</td>
<td>b=T</td>
</tr>
<tr>
<td>c=T</td>
<td>c=F</td>
</tr>
</tbody>
</table>

\[ T \quad F \]
MC/DC Coverage

- A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate

\[ \varphi(a,b,c) \neq \varphi(a,b,\neg c) \]

\[
\begin{array}{ccc}
(a || b && c) \\
\begin{array}{ccc}
a=F & a=F \\
b=T & b=T \\
c=T & c=F \\
\end{array}
\end{array}
\]

This pair of tests shows the impact of \( c \).
MC/DC Coverage

- A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate.

- The basic steps come from `&` and `|`
MC/DC Coverage

- A clause **determines** the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate.

- The basic steps come from & and |

  \[
  a \& b \\
  \text{If } a=\text{True}, \ b \text{ determines the outcome.}
  \]
MC/DC Coverage

- A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate.

- The basic steps come from & and |

\[
\begin{align*}
a \& b \\
\text{If } a=\text{True}, \ b \text{ determines the outcome.}
\end{align*}
\]
\[
\begin{align*}
a \mid b \\
\text{If } a=\text{False}, \ b \text{ determines the outcome.}
\end{align*}
\]
MC/DC Coverage

- A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate.

- The basic steps come from $\&$ and $|$.

  - $a \& b$
    - If $a=$True, $b$ determines the outcome.
  
  - $a \mid b$
    - If $a=$False, $b$ determines the outcome.

- By definition, solve $\varphi_c=$true $\oplus$ $\varphi_c=$false
MC/DC Coverage

- Given $a \mid (b \& c)$, generate tests for $a$
  
  a has impact $\iff$
MC/DC Coverage

- Given $a \mid (b \& c)$, generate tests for $a$

  $a$ has impact $\iff \#T \mid (b \& c) \neq \#F \mid (b \& c)$
MC/DC Coverage

- Given \( a \mid (b \& c) \), generate tests for \( a \)

  \( a \) has impact \( \Leftrightarrow \) \( \#T \mid (b \& c) \neq \#F \mid (b \& c) \)

  \( \Leftrightarrow \) \( \#T \neq b \& c \)
MC/DC Coverage

- Given \( a \mid (b \& c) \), generate tests for \( a \)

\[
\begin{align*}
a \text{ has impact} & \iff \#T \mid (b \& c) \neq \#F \mid (b \& c) \\
& \iff \#T \neq b \& c \\
& \iff \#T = \neg b \mid \neg c
\end{align*}
\]
MC/DC Coverage

- Given \( a \mid (b \land c) \), generate tests for \( a \)

\[
\begin{align*}
\text{a has impact} & \iff \#T \mid (b \land c) \neq \#F \mid (b \land c) \\
& \iff \#T \neq b \land c \\
& \iff \#T = \neg b \mid \neg c \\
& \iff \text{b is false or c is false}
\end{align*}
\]
MC/DC Coverage

- Given \( a \mid (b \& c) \), generate tests for \( a \)

  \[
  \begin{align*}
  a \text{ has impact} & \iff \ #T \mid (b \& c) \neq \ #F \mid (b \& c) \\
  & \iff \ #T \neq b \& c \\
  & \iff \ #T = \neg b \mid \neg c \\
  & \iff b \text{ is false or } c \text{ is false}
  \end{align*}
  \]

defines two different ways to test \( a \)
MC/DC Coverage

- Given \( a \mid (b \& c) \), generate tests for a

  a has impact \( \Leftrightarrow \) \( \#T \mid (b \& c) \neq \#F \mid (b \& c) \)

  \( \Leftrightarrow \) \( \#T \neq b \& c \)

  \( \Leftrightarrow \) \( \#T = \neg b \mid \neg c \)

  \( \Leftrightarrow \) b is false or c is false

defines two different ways to test a

Have b be \( \#F \)

a=\#T, b=\#F, c=\#T
a=\#F, b=\#F, c=\#T
MC/DC Coverage

- Given $a \mid (b \& c)$, generate tests for $a$

  $a$ has impact $\iff$  $\#T \mid (b \& c) \neq \#F \mid (b \& c)$

  $\iff$  $\#T \neq b \& c$

  $\iff$  $\#T = \neg b \mid \neg c$

  $\iff$  $b$ is false or $c$ is false

  defines two different ways to test $a$

  Have $b$ be $\#F$
  
  $a=$#F, $b=#F$, $c=#F$
  $a=$#T, $b=#F$, $c=#T$

  Have $c$ be $\#F$
  
  $a=$#T, $b=#T$, $c=#F$
  $a=$#F, $b=#T$, $c=#F$
MC/DC Coverage

- Given $a \mid (b \& c)$, generate tests for $b$

  $b$ has impact
MC/DC Coverage

- Given $a \mid (b \& c)$, generate tests for $b$

  $b$ has impact
MC/DC Coverage

- Given $a \mid (b \& c)$, generate tests for $b$

  $b$ has impact

$c$ must be true for impact
**MC/DC Coverage**

- Given $a \mid (b \& c)$, generate tests for $b$
  - $b$ has impact
  - $a$ must be false for impact
MC/DC Coverage

- Given $a \mid (b \& c)$, generate tests for $b$
  
  $b$ has impact $\iff a = \#F \& c = \#T$
MC/DC Coverage

- What about \((a \& b) \mid (a \& \neg b)\)?
  - Can you show the impact of \(a\)?
MC/DC Coverage

- What about \((a \& b) \mid (a \& \neg b)\)?
  - Can you show the impact of \(a\)?
  - Can you show the impact of \(b\)?
MC/DC Coverage

- What about \((a \land b) \lor (a \land \neg b)\)?
  - Can you show the impact of \(a\)?
  - Can you show the impact of \(b\)?

Lack of MC/DC coverage can also identify bugs.
MC/DC Coverage

- What about \((a \& b) \mid (a \& \neg b)\)?
  - Can you show the impact of \(a\)?
  - Can you show the impact of \(b\)?

- **BUT NASA recommended** *not generating* MC/DC coverage.
  - Use MC/DC as a means of *evaluating* test suites generated by other means
MC/DC Coverage

- What about \((a \land b) \lor (a \land \neg b)\)?
  - Can you show the impact of \(a\)?
  - Can you show the impact of \(b\)?

- BUT NASA recommended *not generating* MC/DC coverage.
  - Use MC/DC as a means of *evaluating* test suites generated by other means

- In practice there are many pitfalls for getting value out of it
MC/DC Coverage

- What about \((a \& b) \mid (a \& \neg b)\)?
  - Can you show the impact of \(a\)?
  - Can you show the impact of \(b\)?

- BUT NASA recommended *not generating* MC/DC coverage.
  - Use MC/DC as a means of *evaluating* test suites generated by other means

- In practice there are many pitfalls for getting value out of it
  - If you refactor the code, why does the coverage change?
MC/DC Coverage

- What about \((a \& b) \mid (a \& \neg b)\)?
  - Can you show the impact of \(a\)?
  - Can you show the impact of \(b\)?

- BUT NASA recommended *not generating* MC/DC coverage.
  - Use MC/DC as a means of *evaluating* test suites generated by other means

- In practice there are many pitfalls for getting value out of it
  - If you refactor the code, why does the coverage change?
  - How do you deal with short-circuiting operators?
  - ...
Mutation Testing
Mutation Analysis

- Instead of covering program elements, estimate defect finding on a sample of representative bugs
Mutation Analysis

- Instead of covering program elements, estimate defect finding on a sample of representative bugs

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

- Instead of covering program elements, estimate defect finding on a sample of representative bugs

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

- Instead of covering program elements, estimate defect finding on a sample of representative bugs

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

\[
\begin{align*}
  a &= b + c \\
  a &= b \ast c
\end{align*}
\]
Mutation Analysis

- Instead of covering program elements, estimate defect finding on a sample of representative bugs

- **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

- Instead of covering program elements, estimate defect finding on a sample of representative bugs

- **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

- Instead of covering program elements, estimate defect finding on a sample of representative bugs.

- 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

- Instead of covering program elements, estimate defect finding on a sample of representative bugs.

- 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 Analysis

- Instead of covering program elements, estimate defect finding on a sample of representative bugs.

- 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.

Depending on the source, these may swap...
Mutation

- What are possible mutants?

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

• What are possible mutants?

```cpp
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?

```java
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:**
  - *(Still Born)* Not compilable
**Mutation**

- **What are possible mutants?**

  ```c
  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:**
  - *(Still Born)* Not compilable
  - *(Trivial)* Killed by most test cases
Mutation

- What are possible mutants?

```java
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:**
  - *(Still Born)* Not compilable
  - *(Trivial)* Killed by most test cases
  - *(Equivalent)* Indistinguishable from original program
Mutation

- What are possible mutants?
  ```java
  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:
  - *(Still Born)* Not compilable
  - *(Trivial)* Killed by most test cases
  - *(Equivalent)* Indistinguishable from original program
  - *(Redundant)* Indistinguishable from other mutants
Mutation

- What are possible mutants?

```c
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:
  - (Still Born) Not compilable
  - Trivial
  - Equivalent
  - Redundant

Filtering these out is theoretically impossible, yet it is an important & active area of research.
Mutation

```c
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

- Mimic mistakes
- Encode knowledge from other techniques

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;
}
Mutation

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

- Mimic mistakes
- Encode knowledge from other techniques

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

```c
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) {`

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

- Mimic mistakes
- Encode knowledge from other techniques
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;
    minVal = b;
    if (b < a) {
        if (b > a) {
            if (b < minVal) {
                minVal = b;
            }
        }
    }
    return minVal;
}

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

- Mimic mistakes
- Encode knowledge from other techniques
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;
    minVal = b;
    if (b < a) {
        if (b > a) {
            if (b < minVal) {
                minVal = b;
                BOMB();
            }
        }
    }
    return minVal;
}

Mutant 1: minVal = b;
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

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

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

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

- Mimic mistakes
- Encode knowledge from other techniques
int min(int a, int b) {
    int minVal;
    minVal = a;
    if (b < a) {
        minVal = b;
    }
    return minVal;
}

What mimics statement coverage?

- Mimic mistakes
- Encode knowledge from other techniques

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

Mutant 1: minVal = b;
Mutant 2: if (b > a) {
    minVal = b;
}
Mutant 3: if (b < minVal) {
    BOMB();
    minVal = a;
    minVal = failOnZero(b);
}
Mutant 4: BOMB();
Mutant 5: minVal = a;
Mutant 6: minVal = failOnZero(b);
    return minVal;
}
# Mutation Analysis

## Mutants

| Mutant 1  
| Mutant 2  
| Mutant 3  
| Mutant 4  
| Mutant 5  
| Mutant 6  |
# Mutation Analysis

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutant 1</td>
<td>min(1, 2) → 1</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>min(2, 1) → 1</td>
</tr>
<tr>
<td>Mutant 3</td>
<td></td>
</tr>
<tr>
<td>Mutant 4</td>
<td></td>
</tr>
<tr>
<td>Mutant 5</td>
<td></td>
</tr>
<tr>
<td>Mutant 6</td>
<td></td>
</tr>
</tbody>
</table>
Mutation Analysis

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

**Test Suite**

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>min(1,2)</td>
<td>1</td>
</tr>
<tr>
<td>min(2,1)</td>
<td>1</td>
</tr>
</tbody>
</table>

Try every mutant on test 1.
Mutation Analysis

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutant 1</td>
<td>min(1,2) → 1</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>min(2,1) → 1</td>
</tr>
<tr>
<td>Mutant 3</td>
<td></td>
</tr>
<tr>
<td>Mutant 4</td>
<td></td>
</tr>
<tr>
<td>Mutant 5</td>
<td></td>
</tr>
<tr>
<td>Mutant 6</td>
<td></td>
</tr>
</tbody>
</table>

Mutant 1 and Mutant 2 are killed by the test suite.
Try every *live* mutant on test 2.
Mutation Analysis

Mutants

<table>
<thead>
<tr>
<th>Mutant 1</th>
<th>Mutant 2</th>
<th>Mutant 3</th>
<th>Mutant 4</th>
<th>Mutant 5</th>
<th>Mutant 6</th>
</tr>
</thead>
</table>

Test Suite

- \( \text{min}(1,2) \rightarrow 1 \)
- \( \text{min}(2,1) \rightarrow 1 \)
## Mutation Analysis

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutant 1</td>
<td>min(1,2) → 1</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>min(2,1) → 1</td>
</tr>
<tr>
<td>Mutant 3</td>
<td>Killed</td>
</tr>
<tr>
<td>Mutant 4</td>
<td>Killed</td>
</tr>
<tr>
<td>Mutant 5</td>
<td>Killed</td>
</tr>
<tr>
<td>Mutant 6</td>
<td>Killed</td>
</tr>
</tbody>
</table>

So the mutation score is...
So the mutation score is... \textcolor{red}{4/5}. Why?
Mutation Analysis

Mutants

<table>
<thead>
<tr>
<th>Mutant 1</th>
<th>Mutant 2</th>
<th>Mutant 3</th>
<th>Mutant 4</th>
<th>Mutant 5</th>
<th>Mutant 6</th>
</tr>
</thead>
</table>

Test Suite

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Expected Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>min(1,2)</td>
<td>1</td>
</tr>
<tr>
<td>min(2,1)</td>
<td>1</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Mutant 1</th>
<th>Mutant 2</th>
<th>Mutant 3</th>
<th>Mutant 4</th>
<th>Mutant 5</th>
<th>Mutant 6</th>
</tr>
</thead>
</table>

**Test Suite**

- $\text{min}(1, 2) \rightarrow 1$
- $\text{min}(2, 1) \rightarrow 1$

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 < \textbf{minVal})
        minVal = b;
    return minVal;
```

```
int minVal;
minVal = a;
if (b < a)
    minVal = \textbf{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:

\[
\frac{\#\text{Killed}}{\#\text{Mutants}}
\]

Start with the simplest 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:

\[
\text{\#Killed} - \text{\#Killed Duplicates} \\
\text{\#Mutants} - \text{\#Equivalent} - \text{\#Duplicates}
\]

Updated for 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  Infection  Propagation
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.

```c
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 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)
Mutation Testing

- Considered one of the strongest criteria
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)

- Massive number of criteria.

Why?
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)

- Massive number of criteria.
  - The large space of mutants means that current users sample or select
  - But these approaches are known to be less effective
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)

- Massive number of criteria.
  - The large space of mutants means that current users sample or select
  - But these approaches are known to be less effective

- **Scaling up mutation testing is an area of open research**
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)

- Massive number of criteria.
  - The large space of mutants means that current users sample or select
  - But these approaches are known to be less effective

- **Scaling up mutation testing is an area of open research**
  - Better pruning? (equivalent, duplicate, invalid, equivalent WRT test suite)
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)

- Massive number of criteria.
  - The large space of mutants means that current users sample or select
  - But these approaches are known to be less effective

- Scaling up mutation testing is an area of open research
  - Better pruning? (equivalent, duplicate, invalid, equivalent WRT test suite)
  - Identifying subsumption relations (If x is killed, y is also killed) (semantics based, ML based, ...) [Chekam 2020, Kurtz 2014, Just 2014]
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)

- Massive number of criteria.
  - The large space of mutants means that current users sample or select
  - But these approaches are known to be less effective

- Scaling up mutation testing is an area of open research
  - Better pruning? (equivalent, duplicate, invalid, equivalent WRT test suite)
  - Identifying subsumption relations (If x is killed, y is also killed)
    (semantics based, ML based, ...) [Chekam 2020, Kurtz 2014, Just 2014]
  - Better abstractions (source level, IR level, complex faults) [Hariri 2019, Wong 2020]
Mutation Testing

- Considered one of the strongest criteria
  - Mimics some input specifications
  - Mimics some traditional coverage (statement, branch, ...)

- Massive number of criteria.
  - The large space of mutants means that current users sample or select
  - But these approaches are known to be less effective

- **Scaling up mutation testing is an area of open research**
  - Better pruning? (equivalent, duplicate, invalid, equivalent WRT test suite)
  - Identifying *subsumption* relations (If x is killed, y is also killed)
    (semantics based, ML based, ...) [Chekam 2020, Kurtz 2014, Just 2014]
  - Better abstractions (source level, IR level, complex faults) [Hariri 2019, Wong 2020]
  - Better execution strategies (distributed, parallel, maximizing 1 run info)
    [Tokumoto 2016, Gopinath 2016, Just 2014]
Mutation Testing

- **How is it currently used in practice?**
  - Google can integrate results into the code review workflow [Petrovic 2018]
  - Facebook can use ML to guide the mutant process but not widely [Beller 2021]
  - Mutant sampling is still prevalent despite shortcomings [Petrovic 2018]
  - Tools are available across languages, but data for smaller firms is challenging
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.
  – $\text{Covstmt}(T_1) > \text{Covstmt}(T_2) \rightarrow ?$
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - \( \text{Covstmt}(T1) > \text{Covstmt}(T2) \rightarrow ? \)  
    Is T1 more likely to find more bugs?
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $\text{Covstmt}(T_1) > \text{Covstmt}(T_2) \rightarrow ?$
  - Is $T_1$ 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.
  - \( \text{Covstmt}(T_1) > \text{Covstmt}(T_2) \) →

- Understanding the relationships between different levels of coverage and different approaches to coverage is actually challenging & fraught with error [Chen 2020]
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $\text{Covstmt}(T_1) > \text{Covstmt}(T_2) \rightarrow$

- Understanding the relationships between different *levels* of coverage and different *approaches* to coverage is actually challenging & fraught with error [Chen 2020]
  - Having statement/branch coverage is better than not having it
Traditional Coverage vs Mutation

• Statement & branch based coverage are the most popular adequacy measures in practice.
  – \( \text{Covstmt}(T_1) > \text{Covstmt}(T_2) \rightarrow \)

• Understanding the relationships between different levels of coverage and different approaches to coverage is actually challenging & fraught with error [Chen 2020]
  – Having statement/branch coverage is better than not having it
  – Beyond statements/branches, mutation coverage provides better assurance
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - \( \text{Covstmt}(T_1) > \text{Covstmt}(T_2) \rightarrow \)

- Understanding the relationships between different levels of coverage and different approaches to coverage is actually challenging & fraught with error [Chen 2020]
  - Having statement/branch coverage is better than not having it
  - Beyond statements/branches, mutation coverage provides better assurance

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

- *Regression Testing*
Regression Testing

- *Regression Testing*
  - Retesting software as it evolves to ensure previous functionality
Regression Testing

- *Regression Testing*
  - Retesting software as it evolves to ensure previous functionality

- Useful as a tool for *ratcheting* software quality
Regression Testing

- **Regression Testing**
  - Retesting software as it evolves to ensure previous functionality

- Useful as a tool for *ratcheting* software quality

**What is a ratchet?**
Regression Testing

- *Regression Testing*
  - Retesting software as it evolves to ensure previous functionality

- Useful as a tool for *ratcheting* software quality

*What is a ratchet?*
Regression Testing

- *Regression Testing*  
  - Retesting software as it evolves to ensure previous functionality

- Useful as a tool for *ratcheting* software quality

- Regression tests further enable making changes
Why Use Regression Testing

- As software evolves, previously working functionality can fail.
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
  - Changing one component can unintentionally impact another.
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
  - Changing one component can unintentionally impact another.
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
  - Changing one component can unintentionally impact another.
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
  - Changing one component can unintentionally impact another.

```cpp
Header
parseHeader(std::ifstream& in) {
  ...
}
```

```cpp
Contents
parseFile(std::path& p) {
  ...
  auto header = parseHeader(...);
  ...
}
```
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
  - Changing one component can unintentionally impact another.
  - New environments can introduce unexpected behavior in components that originally work.
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
  - Changing one component can unintentionally impact another.
  - New environments can introduce unexpected behavior in components that originally work.

- **Most testing is regression testing** (testing in response to change)
Why Use Regression Testing

- As software evolves, previously working functionality can fail
  - Software is complex & interconnected.
  - Changing one component can unintentionally impact another.
  - New environments can introduce unexpected behavior in components that originally work.

- **Most testing is regression testing**

- **Ensuring previous functionality can require large test suites. Are they always realistic?**
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests

But this is more or less where we started...
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests

- Sometimes not all tests need to run with each commit
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests

- Sometimes not all tests need to run with each commit
  - Run a subset of sanity or smoke tests for commits
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests

- Sometimes not all tests need to run with each commit
  - Run a subset of sanity or *smoke tests* for commits

These mostly validate the build process & core behaviors.
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests

- Sometimes not all tests need to run with each commit
  - Run a subset of sanity or smoke tests for commits
  - Run more thorough tests nightly
Limiting Regression Suites

- Be careful not to add redundant tests to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests

- Sometimes not all tests need to run with each commit
  - Run a subset of sanity or smoke tests for commits
  - Run more thorough tests nightly
  - “ ” weekly
  - “ ” preparing for milestones/ integration
Limiting Regression Suites

- Be careful not to add redundant test to the test suite.
  - Every bug may indicate a useful behavior to test
  - Test adequacy criteria can limit the other tests

- Sometimes not all tests need to run with each commit
  - Run a subset of sanity or smoke tests for commits
  - Run more thorough tests nightly
  - “ ” weekly
  - “ ” preparing for milestones/ integration

- We may further reduce work using information about the change....
Limiting Regression Testing

- Can we be smarter about which test we run & when?

What else could we do?
Limiting Regression Testing

- Can we be smarter about which test we run & when?
- Change Impact Analysis
  - Identify *how* changes affect the rest of software
Limiting Regression Testing

- Can we be smarter about which test we run & when?
- **Change Impact Analysis**
  - Identify how changes affect the rest of software
- Can decide which tests to run on demand
Limiting Regression Testing

- Can we be smarter about which test we run & when?

- **Change Impact Analysis**
  - Identify how changes affect the rest of software

- Can decide which tests to run on demand
  - **Conservative:** run all tests
  - **Cheap:** run tests with test requirements related to the changed lines
Limiting Regression Testing

- Can we be smarter about which test we run & when?
- **Change Impact Analysis**
  - Identify how changes affect the rest of software
- **Can decide which tests to run on demand**
  - **Conservative**: run all tests
  - **Cheap**: run tests with test requirements related to the changed lines

Is the cheap approach *enough*?
Limiting Regression Testing

- Can we be smarter about which test we run & when?
- **Change Impact Analysis**
  - Identify how changes affect the rest of software
- **Can decide which tests to run on demand**
  - **Conservative**: run all tests
  - **Cheap**: run tests with test requirements related to the changed lines
  - **Middle ground**: Run those tests affected by how changes *propagate through* the software?
Limiting Regression Testing

- Can we be smarter about which test we run & when?
- **Change Impact Analysis**
  - Identify how changes affect the rest of software
- Can decide which tests to run on demand
  - **Conservative**: run all tests
  - **Cheap**: run tests with test requirements related to the changed lines
  - **Middle ground**: Run those tests affected by how changes *propagate* through the software?

In practice, tools can assist in finding out which tests need to be run.
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
  - The analysis detects *dependencies* between *components*

```cpp
Header
parseHeader(std::ifstream& in) {
  ...
}

Contents
parseFile(std::path& p) {
  ...
  auto header = parseHeader(...);
  ...
}
```
Given a set of changes, regression test selection determines which tests to execute
- The analysis detects dependencies between components
- Only tests for components (transitively) dependent on a change need to run

```cpp
// Header
parseHeader(std::ifstream& in) {
  ... 
}

// Contents
parseFile(std::path& p) {
  ...
  auto header = parseHeader(...);
  ...
}
```
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
  - The analysis detects dependencies between components
  - Only tests for components (transitively) dependent on a change need to run
  - Different forms of dependence impact the efficiency, safety, & reduction
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
  - The analysis detects dependencies between components
  - Only tests for components (transitively) dependent on a change need to run
  - Different *forms* of dependence impact the *efficiency, safety, & reduction*
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
  - The analysis detects dependencies between components
  - Only tests for components (transitively) dependent on a change need to run
  - Different forms of dependence impact the efficiency, safety, & reduction

- The granularity of the analysis also affects all aspects of performance
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
  - The analysis detects dependencies between components
  - Only tests for components (transitively) dependent on a change need to run
  - Different forms of dependence impact the efficiency, safety, & reduction

- The granularity of the analysis also affects all aspects of performance

Why might *project* dependencies be too conservative?
Given a set of changes, regression test selection determines which tests to execute:
- The analysis detects dependencies between components
- Only tests for components (transitively) dependent on a change need to run
- Different forms of dependence impact the efficiency, safety, & reduction

The granularity of the analysis also affects all aspects of performance.

Why might class dependencies be too conservative?
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
  - The analysis detects dependencies between components
  - Only tests for components (transitively) dependent on a change need to run
  - Different forms of dependence impact the efficiency, safety, & reduction

- The granularity of the analysis also affects all aspects of performance

Why might *function* dependencies be too conservative?
Change Impact Analysis & Regression Test Selection

- Given a set of changes, regression test selection determines which tests to execute
  - The analysis detects dependencies between components
  - Only tests for components (transitively) dependent on a change need to run
  - Different forms of dependence impact the efficiency, safety, & reduction

- The granularity of the analysis also affects all aspects of performance

- We will discuss the techniques underneath this as static & dynamic program analysis
Additional Strategies for Speeding Up Testing

- **Test Case Prioritization**
  - Can we run the tests in an order such that the suite fails faster? [Elbaum 2002]
Additional Strategies for Speeding Up Testing

- **Test Case Prioritization**
  - Can we run the tests in an order such that the suite fails faster? [Elbaum 2002]

- **Test Suite Reduction**
  - Can we shrink our test suite but still test enough?
  - Current evidence points to test suite reduction performing poorly in practice. [Shi 2018]
Additional Strategies for Speeding Up Testing

- **Test Case Prioritization**
  - Can we run the tests in an order such that the suite fails faster? [Elbaum 2002]

- **Test Suite Reduction**
  - Can we shrink our test suite but still test enough?
  - Current evidence points to test suite reduction performing poorly in practice. [Shi 2018]

- **Bug Prediction**
  - Can we mine properties of a repository to predict where bugs will likely be?
  - Evidence indicated a mismatch between techniques & outcomes [Lewis 2013]
  - But advances are ongoing [Nam 2017]
Using Test Suites For Other Purposes
Leveraging Test Suites Further

- We have considered how to
  - write tests well.
  - measure & assess a test suite.
  - efficiently & effectively add testing into a workflow.
Leveraging Test Suites Further

- We have considered how to
  - write tests well.
  - measure & assess a test suite.
  - efficiently & effectively add testing into a workflow.

- All of these aid using tests to know *when bugs occur*
Leveraging Test Suites Further

- We have considered how to
  - write tests well.
  - measure & assess a test suite.
  - efficiently & effectively add testing into a workflow.

- All of these aid using tests to know *when bugs occur*

- **But we often care about other tasks:**
  - *Investigating* why a bug exists
  - *Repairing* a bug
  - *Hardening* a program against attack
  - *Reusing* old software (even if the source code has been lost)
Leveraging Test Suites Further

- We have considered how to
  - write tests well.
  - measure & assess a test suite.
  - efficiently & effectively add testing into a workflow.

- All of these aid using tests to know *when bugs occur*

- But we often care about other tasks:
  - *Investigating* why a bug exists
  - *Repairing* a bug
  - *Hardening* a program against attack
  - *Reusing* old software (even if the source code has been lost)

- All of these can be aided, guided, or automated using test suites
Leveraging Test Suites Further

- What information does a test suite give us?
Leveraging Test Suites Further

- What information does a test suite give us?
  - A weak *black box oracle* for program correctness

Is foo.out correct?
Leveraging Test Suites Further

- What information does a test suite give us?
  - A weak black box oracle for program correctness
  - Observable information about program behavior during tests
Leveraging Test Suites Further

- What information does a test suite give us?
  - A weak black box oracle for program correctness
  - Observable information about program behavior during tests

- We can run a test suite (even in a loop) to build tasks using these tools!
Leveraging Test Suites Further

- What information does a test suite give us?
  - A weak black box oracle for program correctness
  - Observable information about program behavior during tests

- We can run a test suite (even in a loop) to build tasks using these tools!

- **Interesting questions:**
  - What occurs in tests that pass?
  - What occurs in tests that fail?
  - Can I search for X that is part of a correct program?
  - Can I search for X that is part of a buggy program?
  - ...
Fault Localization

- Suppose that a bug at a statement causes some tests to fail
Fault Localization

- Suppose that a bug at a statement causes some tests to fail
  - The test suite embeds information that can aid our search for the bug
  - *Fault localization* ranks the locations in a program to consider
Fault Localization

• Suppose that a bug at a statement causes some tests to fail
  – The test suite embeds information that can aid our search for the bug
  – *Fault localization* ranks the locations in a program to consider

• Given
  – A test suite $T = \langle \{t_i\}, o \rangle$
  – *Passing* tests $p \subseteq \{t_i\}$
  – *Failing* tests $f \subseteq \{t_i\}$
  – Observable criteria $c(t_i) = c_i$
Fault Localization

- Suppose that a bug at a statement causes some tests to fail
  - The test suite embeds information that can aid our search for the bug
  - *Fault localization* ranks the locations in a program to consider

- **Given**
  - A test suite $T = \langle \{ti\}, o \rangle$
  - Passing tests $p \subseteq \{ti\}$
  - Failing tests $f \subseteq \{ti\}$
  - Observable criteria $c(t_i) = c_i$

- **Produce**
  - A ranked list of *locations* $[li]$ for a developer to consider
Fault Localization

- Suppose that a bug at a statement causes some tests to fail.
- The test suite embeds information that can aid our search for the bug.
- Fault localization ranks the locations in a program to consider.

- Given:
  - A test suite \( T = \{t_i, o\} \)
  - Passing tests \( p \subset \{t_i\} \)
  - Failing tests \( f \subset \{t_i\} \)
  - Observable criteria \( c(t_i) = c_i \)

- Produce
  - A ranked list of locations \([l_i]\) for a developer to consider.

```python
if condition:
    x = a + b
else:
    y = c * d
return x + y
```
Fault Localization

- Suppose that a bug at a statement causes some tests to fail.
  - The test suite embeds information that can aid our search for the bug.
  - Fault localization ranks the locations in a program to consider.

- Given:
  - A test suite $T = \langle \{t_i\}, o \rangle$
  - Passing tests $p \subseteq \{t_i\}$
  - Failing tests $f \subseteq \{t_i\}$
  - Observable criteria $c(t_i) = c_i$

- Produce:
  - A ranked list of locations $[l_i]$ for a developer to consider.

```python
if condition:
    x = a + b
else:
    y = c * d
return x + y
```
Fault Localization

- Suppose that a bug at a statement causes some tests to fail
  - The test suite embeds information that can aid our search for the bug
  - *Fault localization* ranks the locations in a program to consider

- Given
  - A test suite $T = \langle \{t_i\}, o \rangle$
  - Passing tests $p \subset \{t_i\}$
  - Failing tests $f \subset \{t_i\}$
  - Observable criteria $c(t_i) = c_i$

- Produce
  - A ranked list of locations $[l_i]$ for a developer to consider

- Measures
  - Top-1 is ideal. Outside of Top-10 is not useful for *manual* analysis.
Fault Localization

- Suppose that a bug at a statement causes some tests to fail
  - The test suite embeds information that can aid our search for the bug
  - **Fault localization** ranks the locations in a program to consider

- **Given**
  - A test suite $T = \langle \{t_i\}, o \rangle$
  - Passing tests $p \subseteq \{t_i\}$
  - Failing tests $f \subseteq \{t_i\}$
  - Observable criteria $c(t_i) = c_i$

- **Produce**
  - A ranked list of locations $[l_i]$ for a developer to consider

- **Measures**
  - Top-1 is ideal. Outside of Top-10 is not useful for manual analysis.
Fault Localization

- What criteria might be useful?
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
  - Text comments from code executed by tests
 Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
  - Text comments from code executed by tests
  - ...

  Defining criteria well is an important part of a technique
Fault Localization

• What criteria might be useful?
  – Simple test coverage/adequacy information
  – Important values (return values, arguments, specific functions)
  – Invariants & likely invariants
  – Text comments from code executed by tests
  – ...
    Defining criteria well is an important part of a technique

• A lot of classic techniques focus on, e.g., statement coverage
Fault Localization

- Simple test coverage/adequacy information
- Important values (return values, arguments, specific functions)
- Invariants & likely invariants
- Text comments from code executed by tests

Defining criteria well is an important part of a technique.

A lot of classic techniques focus on, e.g., statement coverage.
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
  - Text comments from code executed by tests
  - ...

Defining criteria well is an important part of a technique.

A lot of classic techniques focus on, e.g., statement coverage

```python
if condition:
    x = a + b
else:
    y = c * d
return x + y
```
Fault Localization

What criteria might be useful?

- Simple test coverage/adequacy information
- Important values (return values, arguments, specific functions)
- Invariants & likely invariants
- Text comments from code executed by tests
- ...

Defining criteria well is an important part of a technique.

A lot of classic techniques focus on, e.g., statement coverage.

```python
if condition:
    x = a + b
else:
    y = c * d
return x + y
```
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
  - Text comments from code executed by tests

Defining criteria well is an important part of a technique.

A lot of classic techniques focus on, e.g., statement coverage.

```python
if condition:
    x = a + b
else:
    y = c * d
return x + y
```
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
  - Text comments from code executed by tests
  - ...

Defining criteria well is an important part of a technique.

A lot of classic techniques focus on, e.g., statement coverage.

What does your intuition tell you about likely causes for the bug?
What criteria might be useful?

- Simple test coverage/adequacy information
- Important values (return values, arguments, specific functions)
- Invariants & likely invariants
- Text comments from code executed by tests
- ...

Defining criteria well is an important part of a technique.

A lot of classic techniques focus on, e.g., statement coverage.

What does your intuition tell you about likely causes for the bug?
Fault Localization

● What criteria might be useful?
  – Simple test coverage/adequacy information
  – Important values (return values, arguments, specific functions)
  – Invariants & likely invariants
  – Text comments from code executed by tests

Defining criteria well is an important part of a technique

● A lot of classic techniques focus on, e.g., statement coverage

● How should we prioritize?

```python
if condition:
    x = a + b
else:
    y = c * d
return x + y
```
Fault Localization

What criteria might be useful?

- Simple test coverage/adequacy information
- Important values (return values, arguments, specific functions)
- Invariants & likely invariants
- Text comments from code executed by tests

Defining criteria well is an important part of a technique

A lot of classic techniques focus on, e.g., statement coverage

How should we prioritize?

- Heuristic [Jones 2005, Jiang 2019]

```python
if condition:
x = a + b
else:
y = c * d
return x + y
```
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
  - Text comments from code executed by tests
  -...

- Defining criteria well is an important part of a technique

- A lot of classic techniques focus on, e.g., statement coverage

- How should we prioritize?
  - Heuristic [Jones 2005, Jiang 2019]
## Fault Localization

### What criteria might be useful?

- Simple test coverage/adequacy information
- Important values (return values, arguments, specific functions)
- Invariants & likely invariants
- Text comments from code executed by tests
- ...

Defining criteria well is an important part of a technique.

### A lot of classic techniques focus on, e.g., statement coverage

### How should we prioritize?

- **Heuristic** [Jones 2005, Jiang 2019]
- **Statistical** [Landberg 2018]

```python
if condition:
    x = a + b
else:
    y = c * d
return x + y
```

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

\[
P(H_i | u_i) \quad \text{Likelihood that } u_i \text{ caused the failure when executed} \]

\[
\frac{\text{failed}(s)/\text{totalfailed}}{\text{failed}(s)/\text{totalfailed} + \text{passed}(s)/\text{totalpassed}}
\]

[Tarantula]

[Doric]
Fault Localization

- What criteria might be useful?
  - Simple test coverage/adequacy information
  - Important values (return values, arguments, specific functions)
  - Invariants & likely invariants
  - Text comments from code executed by tests

Defining criteria well is an important part of a technique

- A lot of classic techniques focus on, e.g., statement coverage

**How should we prioritize?**
- Heuristic [Jones 2005, Jiang 2019]
- Statistical [Landberg 2018]
- ML based [Li 2019]
- Hybrid models [Zou 2019]
- ...

if condition:
    x = a + b
else:
    y = c * d
return x + y

\[
\begin{array}{cccc}
\text{T1} & \text{T2} & \text{T3} & \text{T4} \\
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\end{array}
\]

\[
\frac{\text{failed}(s)}{\text{total} \text{failed}} + \frac{\text{passed}(s)}{\text{total} \text{passed}}
\]

\[
P(H_i|u_i) \quad \text{Likelihood that } u_i \text{ caused the failure when executed}
\]

[Tarantula]

[Doric]
Fault Localization

- The state of the art is pushed further each year
  - \(~50\% @ \text{Top-1 in 2019}\) [Zou 2019]
  - \(~76\% @ \text{Top-10 in 2019}\)
  - (On standard benchmarks)
Fault Localization

- The state of the art is pushed further each year
  - ~50% @ Top-1 in 2019 [Zou 2019]
  - ~76% @ Top-10 in 2019
  - (On standard benchmarks)

- Still challenges remain
Fault Localization

- The state of the art is pushed further each year
  - ~50% @ Top-1 in 2019 [Zou 2019]
  - ~76% @ Top-10 in 2019
  - (On standard benchmarks)

- Still challenges remain
  - Is localization the main task in debugging?
The state of the art is pushed further each year
- ~50% @ Top-1 in 2019 [Zou 2019]
- ~76% @ Top-10 in 2019
- (On standard benchmarks)

Still challenges remain
- Is localization the main task in debugging?
- Suppose you do localize, what next?
  Understanding [Parnin 2011]
  Fixing
  Assessing
Fault Localization

- The state of the art is pushed further each year
  - ~50% @ Top-1 in 2019 [Zou 2019]
  - ~76% @ Top-10 in 2019
  - (On standard benchmarks)

- Still challenges remain
  - Is localization the main task in debugging?
  - Suppose you do localize, what next?
    - Understanding [Parnin 2011]
    - Fixing
    - Assessing

- Perhaps we can push this further....
Automated Program Repair

- **Given**
  - A program P
  - A test suite T
  - Results from localization: [li]
Automated Program Repair

• Given
  – A program P
  – A test suite T
  – Results from localization: \([l_i]\)

• Produce
  – A ranked list of patches/diffs \([\delta_i]\) that make T pass
Automated Program Repair

- **Given**
  - A program \( P \)
  - A test suite \( T \)
  - Results from localization: \( [l_i] \)

- **Produce**
  - A ranked list of patches/diffs \( [\delta_i] \) that make \( T \) pass

- **If we can define a way to \textit{explore} the space of patches, we can use the test suite to \textit{check} the patches!**
Automated Program Repair

- Given
  - A program P
  - A test suite T
  - Results from localization: [li]

- Produce
  - A ranked list of patches/diffs [δi] that make T pass

- If we can define a way to *explore* the space of patches, we can use the test suite to *check* the patches!

```
loop:
    patch = generatePatch()
    if apply(patch,P) passes T:
        return patch
```
Automated Program Repair

• Given
  – A program P
  – A test suite T
  – Results from localization: [li]

• Produce
  – A ranked list of patches/diffs [δi] that make T pass

• If we can define a way to *explore* the space of patches, we can use the test suite to *check* the patches!

• For a given possibly buggy location
  – Enumerative search
  – Constraint guided search
  – ML (e.g. sequence-to-sequence)
Automated Program Repair

- So why isn’t this deployed everywhere?
So why isn’t this deployed everywhere?
- The techniques are still evolving & bleeding edge [CACM 2019]
- Making a test suite pass is not the same as fixing a bug [Durieux 2019, Long 2016, Long 2015]
Automated Program Repair

- So why isn’t this deployed everywhere?
  - The techniques are still evolving & bleeding edge [CACM 2019]
  - Making a test suite pass is not the same as fixing a bug [Durieux 2019, Long 2016, Long 2015]

- Incorporating ML has improved advancements over the last few years, but more advances are needed for broad usability & adoption
Automated Program Repair

- So why isn’t this deployed everywhere?
  - The techniques are still evolving & bleeding edge [CACM 2019]
  - Making a test suite pass is not the same as fixing a bug [Durieux 2019, Long 2016, Long 2015]

- Incorporating ML has improved advancements over the last few years, but more advances are needed for broad usability & adoption

- But... it is now a part of the possible workflow at big companies
  - Google
  - Microsoft
  - Facebook
  - Bloomberg
  - Samsung
  - ...
Testing Challenging Software
Revisiting the Oracle Problem

- When oracles are challenging, testing is challenging
Revisiting the Oracle Problem

- When oracles are challenging, testing is challenging
  - Compilers?
  - Embedded Systems?
  - Graphics drivers?
  - Machine learning?
  - Simulations & Modeling?
Revisiting the Oracle Problem

- When oracles are challenging, testing is challenging
  - Compilers?
  - Embedded Systems?
  - Graphics drivers?
  - Machine learning?
  - Simulations & Modeling?

How would you test software for modeling Covid-19?
Revisiting the Oracle Problem

- When oracles are challenging, testing is challenging
  - Compilers?
  - Embedded Systems?
  - Graphics drivers?
  - Machine learning?
  - Simulations & Modeling?

- Even if we can test specific cases, how much confidence do those cases provide?
Revisiting the Oracle Problem

- When oracles are challenging, testing is challenging
  - Compilers?
  - Embedded Systems?
  - Graphics drivers?
  - Machine learning?
  - Simulations & Modeling?

- Even if we can test *specific* cases, how much confidence do those cases provide?
  - Why is writing oracles hard?
    - The input spaces are often vast & complex.
  - A test suite is unlikely to expose specific pathological combinations.
Revisiting the Oracle Problem

- When oracles are challenging, testing is challenging
  - Compilers?
  - Embedded Systems?
  - Graphics drivers?
  - Machine learning?
  - Simulations & Modeling?

- Even if we can test specific cases, how much confidence do those cases provide?
  - Why is writing oracles hard?
    - The input spaces are often vast & complex.
  - A test suite is unlikely to expose specific pathological combinations.

- We again need additional leverage
  - Additional implementations?
  - Knowledge about the domain
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging

- But mainstream languages tend to have multiple implementations
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use** differential testing
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use *differential testing***
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use *differential testing***
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use differential testing**
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use *differential testing*
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use** differential testing
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use differential testing**

```
Foo.c
```

```
MSVC
foo.out
```

```
GCC
foo.out
```

```
ICC
foo.out
```

How might we test them here?
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use *differential testing*
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use *differential testing*
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use differential testing**
  - Given compilers c1, c2, ..., ci
  - Provide an input I to each and determine correctness by the majority
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use differential testing**
  - Given programs c1, c2, ..., ci
  - Provide an input I to each and determine correctness by the majority
  - Generate many inputs I and assess automatically
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- **Big picture: use differential testing**
  - Given programs $c_1, c_2, \ldots, c_i$
  - Provide an input $I$ to each and determine correctness by the majority
  - Generate many inputs $I$ and assess automatically
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use **differential testing**
  - Given programs c1, c2, ..., ci
  - Provide an input I to each and determine correctness by the majority
  - Generate many inputs I and assess automatically
- To do that, we need to be very careful with our input generator
  - Programs should produce the same results: [Yang 2011]
    - deterministic
    - well defined
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use differential testing
  
  ```c
  printf("%p", &someVariable);
  ```
  - Provide an input I to each and determine correctness by the majority
  - Generate many inputs I and assess automatically
- To do that, we need to be very careful with our input generator
  - Programs should produce the same results: [Yang 2011] deterministic well defined
How Would You Test a Compiler?

- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use differential testing
  - Provide an input I to each and determine correctness by the majority
  - Generate many inputs I and assess automatically
- To do that, we need to be very careful with our input generator
  - Programs should produce the same results: [Yang 2011]
    deterministic
    well defined

```c
int x = INT_MAX;
x = x + 1;
```
Many compiler bugs come from “middle end” optimizations. Complex interactions from multiple rules make testing challenging. But mainstream languages tend to have multiple implementations. Big picture: use differential testing – Provide an input I to each and determine correctness by the majority. Generate many inputs I and assess automatically. To do that, we need to be very careful with our inputs. Programs should produce the same results: deterministic, well defined.

```c
int x = INT_MAX;
x = x + 1;
printf("%p", &someVariable);
```

```c
int x = 5;
while (x) {
    if (x%2) {
        x = x + 1;
    } else {
        x = x - 1;
    }
}
printf("%d", x);
```
Pressing Further

- Is there a way to get more value out of each generated test?
Pressing Further

- Is there a way to get more value out of each generated test?

- **Given**
  - Some test $T$ with oracle $O$
  - Can we produce many more tests?
Pressing Further

- Is there a way to get more value out of each generated test?
- Given
  - Some test $T$ with oracle $O$
  - Can we produce many more tests?
- **Metamorphic Testing**
  - We can generate new tests using the known behavior of existing tests
Pressing Further

- Is there a way to get more value out of each generated test?

- Given
  - Some test $T$ with oracle $O$
  - Can we produce many more tests?

- **Metamorphic Testing**
  - We can generate new tests using the known behavior of existing tests

- **Given**
  - a sequence of tests $T = \{(I_1, O_1), (I_2, O_2), ..., (I_n, O_n)\}$
Pressing Further

- Is there a way to get more value out of each generated test?

- Given
  - Some test T with oracle O
  - Can we produce many more tests?

- Metamorphic Testing
  - We can generate new tests using the known behavior of existing tests

- Given
  - a sequence of tests \( T = \{(I_1, O_1), (I_2, O_2), \ldots, (I_n, O_n)\} \)

- Produce
  - a test \( T_{n+1} = F(\{I_1, I_2, \ldots, I_n\}, \{O_1, O_2, \ldots, O_n\}) \)
Pressing Further

- Is there a way to get more value out of each generated test?

- Given
  - Some test $T$ with oracle $O$
  - Can we produce many more tests?

- Metamorphic Testing
  - We can generate new tests using the known behavior of existing tests

- Given
  - a sequence of tests $T = \{(I_1,O_1), (I_2,O_2), ..., (I_n,O_n)\}$

- Produce
  - a test $T_{n+1} = F(\{I_1, I_2, ..., I_n\}, \{O_1, O_2, ..., O_n\})$

- How might this fit into the compiler test cases?
Metamorphic Testing for Compilers

- There are a large number of ways to change a program without changing its meaning!

[emi project, Le 2014, Sun 2016]
There are a large number of ways to change a program without changing its meaning! [emi project, Le 2014, Sun 2016]
Metamorphic Testing for Compilers

- There are a large number of ways to change a program without changing its meaning! [emi project, Le 2014, Sun 2016]
Metamorphic Testing for Compilers

- There are a large number of ways to change a program without changing its meaning! [emi project, Le 2014, Sun 2016]

- This may seem simple, but it provides a great deal of value today
  - GCC, Clang, MSVC, ICC
  - Vulcan & OpenGL shaders
  - ...

```c
... if (false) {
  ...
}
...

// x is profiled as < 0
... if (x > 0) {
  ...
}
...```
Other Examples of Metamorphic Testing

- Android apps have complex life cycles and often experience UI glitches
Other Examples of Metamorphic Testing

- Android apps have complex life cycles and often experience UI glitches
  - Events come in from the framework
  - Apps need to respond consistently & intuitively

Simplified Activity Lifecycle
[developer.android.com]
Other Examples of Metamorphic Testing

- Android apps have complex life cycles and often event-driven:
  - Events come in from the framework
  - Apps need to respond consistently & intuitively

---

Simplified Activity Lifecycle
[developer.android.com]

Fragment/Activity Lifecycle
[Pomeroy 2014]
Other Examples of Metamorphic Testing

- Android apps have complex life cycles and often experience UI glitches
  - Events come in from the framework
  - Apps need to respond consistently & intuitively
  - Handling events poorly leads to:
    - Crashes
    - Non-responsiveness
    - Unexpected UI changes
    - ...
Other Examples of Metamorphic Testing

- Android apps have complex life cycles and often experience UI glitches
  - Events come in from the framework
  - Apps need to respond consistently & intuitively
  - Handling events poorly leads to:
    - Crashes
    - Non-responsiveness
    - Unexpected UI changes
    - ...

Again, metamorphic testing makes this simpler. Ideas?
Other Examples of Metamorphic Testing

• Android apps have complex life cycles and often experience UI glitches
  – Events come in from the framework
  – Apps need to respond consistently & intuitively
  – Handling events poorly leads to:
    • Crashes
    • Non-responsiveness
    • Unexpected UI changes
    • ...

• Adversarial event sequences can just be injected into existing tests.
  – e.g. Pause-Resume, Pause-Stop-Restart, Rotate-Unrotate, ...
  – Robust behavior is the same with or without these additions [Quist 2015]
Other Examples of Metamorphic Testing

Adversarial event sequences can just be injected into existing tests.
- e.g. Pause-Resume, Pause-Stop-Restart, Rotate-Unrotate, ...
- Robust behavior is the same with or without these additions [Quist 2015]
Other Examples of Metamorphic Testing

- Adversarial event sequences can just be injected into existing tests.
  - e.g. Pause-Resume, Pause-Stop-Restart, Rotate-Unrotate, ...
  - Robust behavior is the same with or without these additions [Quist 2015]
Other Examples of Metamorphic Testing

- Adversarial event sequences can just be injected into existing tests.
  - e.g. Pause-Resume, Pause-Stop-Restart, Rotate-Unrotate, ...
  - Robust behavior is the same with or without these additions [Quist 2015]
Other Examples of Metamorphic Testing

- Adversarial event sequences can just be injected into existing tests.
  - e.g. Pause-Resume, Pause-Stop-Restart, Rotate-Unrotate, ...
  - Robust behavior is the same with or without these additions [Quist 2015]
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - Sentiment analysis extracts & quantifies affective state (opinion)
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)

  “I like the movie” expresses a mild positive opinion
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - Sentiment analysis extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
- A single test case can be modified to create a *family* of other tests exploring known relationships relative to an original test
  - Negation of meaning
  - Relative magnitude
  - Equivalence
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
- A single test case can be modified to create a family of other tests exploring known relationships relative to an original test
  - Negation of meaning
  - Relative magnitude
  - Equivalence

I really <liked> the flight
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
- A single test case can be modified to create a family of other tests exploring known relationships relative to an original test
  - Negation of meaning
  - Relative magnitude
  - Equivalence

  I really <liked> the flight
  enjoyed liked loved regret
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
- A single test case can be modified to create a family of other tests exploring known relationships relative to an original test
  - Negation of meaning
  - Relative magnitude
  - Equivalence

\[
\begin{align*}
\text{I really} & \quad \text{<liked>} & \quad \text{the flight} \\
\text{enjoyed} & \\
\text{liked} & \\
\text{loved} & \\
\text{regret} & \\
\end{align*}
\]
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
- A single test case can be modified to create a family of other tests exploring known relationships relative to an original test
  - Negation of meaning
  - Relative magnitude
  - Equivalence

I really <liked> the flight
enjoyed
liked
loved
regret
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
- A single test case can be modified to create a family of other tests exploring known relationships relative to an original test
  - Negation of meaning
  - Relative magnitude
  - Equivalence

I really <liked> the flight enjoyed liked loved regret
Other Examples of Metamorphic Testing

- Machine learning can be notoriously fickle & challenging to test
- Consider NLP, simple changes to sentences violate expectations
  - *Sentiment analysis* extracts & quantifies affective state (opinion)
  - Does “I like the movie” mean the same as “I do not like the movie”
- A single test case can be modified to create a family of other tests exploring known relationships relative to an original test
  - Negation of meaning
  - Relative magnitude
  - Equivalence

Why isn’t *Santa Claus* in jail?
Why isn’t *the Tooth Fairy* in jail?
Other Examples of Metamorphic Testing

• Machine learning can be notoriously fickle & challenging to test

• Consider NLP, simple changes to sentences violate expectations
  – Sentiment analysis extracts & quantifies affective state (opinion)
  – Does “I like the movie” mean the same as “I do not like the movie”

• A single test case can be modified to create a family of other tests exploring known relationships relative to an original test
  – Negation of meaning
  – Relative magnitude
  – Equivalence

• Basic metamorphic testing tripled the bug discovery rate of ML testers. [Ribeiro 2020]
Summary

- We have seen how to perform standard testing tasks
  - *Constructing* individual tests
  - *Measuring* whether you are testing well
  - *Managing* testing over software evolution
Summary

- We have seen how to perform standard testing tasks
  - Constructing individual tests
  - Measuring whether you are testing well
  - Managing testing over software evolution

- We have seen how to address challenging to test systems
  - Differential & metamorphic testing provide some guidance
Summary

- We have seen how to perform standard testing tasks
  - Constructing individual tests
  - Measuring whether you are testing well
  - Managing testing over software evolution

- We have seen how to address challenging to test systems
  - Differential & metamorphic testing provide some guidance

- We have seen how test suites can be leveraged for further value
  - Localization
  - Repair
  - There are many more opportunities, too!