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

Input ➔ Program ➔ Observed Behavior
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

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

- Test 1: Input
- Test 2: Input
- Test 3: Input
- Test 4: Input
- Test 5: Input
- Test 6: Input
- Test 7: Input

Oracle

Outcome
Software Testing

- The most common way of measuring and ensuring program correctness

Key Issues
- Test suite adequacy
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

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

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

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

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
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

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

```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;
    ...
}
```

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

- **Common structure**

- **Tests should run in isolation**

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

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

```c
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

```
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

```c++
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/unkown 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/unkown
  - White box testing – program structure & semantics can be used
    - symbolic execution
    - call chain synthesis
    - 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

- 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)
  - $\text{turn}(360, \text{direction}) = \text{direction}$
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)
  - $\text{turn}(360, \text{direction}) = \text{direction}$
  - $\text{program}1(x) = \text{program}2(x)$

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)
  - $\text{turn}(360, \text{direction}) = \text{direction}$
  - $\text{program1}(x) = \text{program2}(x)$

- 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?

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

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

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

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

```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

```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?

- 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?

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')
```

<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>
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')
```

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

- 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?

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

```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*

```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?

```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*

How many *injected bugs* can be detected by 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*

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

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
  - ...

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')
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>#T</th>
<th>#F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>#F</td>
<td>abT abcT abcF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>abT</td>
<td>acT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>acF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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?

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

a

#T

#F

c

p

abT
abcT
abCF
acT
acF
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.
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.
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.

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

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

T F  T F
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.

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

```python
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.

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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):
```

```
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.

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):
```

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

Minimum of 2 tests

- a=true, b=true, c=false, d=false
- a=false, b=false, c=true, d=true
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) | T | F | T | T | T | F | F | T | F |
|-------------------------|---|---|---|---|---|---|---|---|---|
| Minimum of 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
  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

- 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 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

\[ \varphi(a, b, c) \neq \varphi(a, b, \neg c) \]
A clause *determines* the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate

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

\[ (a \mid\mid b \&\& c) \]
A clause **determines** the outcome of a predicate when changing only the value of that clause changes the outcome of the predicate

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

\[ (a \lor b \land c) \]

\[
\begin{array}{ccc}
a&b&c \\
F&T&T \\
T&&T \\
\end{array}
\]

\[ T \]
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>
</tbody>
</table>

\[ \begin{array}{cc}
\hline
| c=T | c=F |
\hline
| T   | F   |
\end{array} \]
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

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

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

<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></td>
<td>T</td>
</tr>
<tr>
<td></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

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

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

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

| T    | F   |

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 |

  \[ a \& b \]
  If \( a = \text{True} \), \( b \) 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 |

\[
\begin{align*}
\text{a & b} & \quad \text{If } a=\text{True, } b \text{ determines the outcome.} \\
\text{a | b} & \quad \text{If } a=\text{False, } b \text{ determines the outcome.}
\end{align*}
\]
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 |

\[
\begin{align*}
\text{a} & \& \text{b} \\
\text{If a=} \text{True, b determines the outcome.}
\end{align*}
\]

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

- By definition, solve $\phi_c=\text{true} \, \oplus \, \phi_c=\text{false}$
MC/DC Coverage

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

• Given $a | (b \& c)$, generate tests for $a$

  $a$ has impact $\iff$ 

  $\#T | (b \& c) \neq \#F | (b \& c)$
Given $a \mid (b \land c)$, generate tests for $a$

- $a$ has impact $\Leftrightarrow$ \#T $\mid (b \land c)$ $\neq$ \#F $\mid (b \land c)$
- $\Leftrightarrow$ \#T $\neq$ $b \land c$
MC/DC Coverage

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

  \[
  \begin{align*}
  a \text{ has impact} & \iff \#T \mid (b \& c) \neq \#F \mid (b \& c) \\
  & \iff \#T \neq b \& c \\
  & \iff \#T = \lnot b \mid \lnot c
  \end{align*}
  \]
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
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 $=$ −b $\mid$ −c

  $\iff$ b is false or c is false

  defines two different ways to test a
MC/DC Coverage

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

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

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 $\#F$
  
  Have $c$ be $\#F$

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

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

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

  $a=\#F, b=\#T, c=\#F$
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 \((a \& b) \mid (a \& \neg b)\)?
  - 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 \((a \land b) \mid (a \land \neg b)\)?
  – 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
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

```
  a = b + c  \rightarrow  a = b * c
```
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?

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

```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.

- 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

```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
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;
    Mutant 1: minVal = b;
    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
Mutation

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

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

– Mimic mistakes
– Encode knowledge from other techniques
## Mutations

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

- **Mutant 1:** `minVal = b;`
- **Mutant 2:** `if (b > a) {` (Code change)
- **Mutant 3:** `if (b < minVal) {` (Code change)
- **Mutant 4:** `BOMB();` (Code change)

- **Mimic mistakes**
- **Encode knowledge from other techniques**
```c
int min(int a, int b) {
    int 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;
}

- 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;
    if (b < a) {
        minVal = b;
    }
    return minVal;
}

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

<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>

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>Killed</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**

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutant 1</td>
<td>( \min(1,2) \rightarrow 1 )</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>( \min(2,1) \rightarrow 1 )</td>
</tr>
<tr>
<td>Mutant 3</td>
<td>Killed</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>

Try every *live* mutant on test 2.
# 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>
Mutation Analysis

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

Test Suite
\[ \text{min}(1, 2) \rightarrow 1 \]
\[ \text{min}(2, 1) \rightarrow 1 \]

So the mutation score is...
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?
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></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

<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... **4/5. Why?**

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

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

Equivalent to the original! There is no injected bug.
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:

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

Updated for modern handling of duplicate & equivalent mutants
Equivalent Mutants

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

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

- Detecting equivalent mutants is *undecidable* in general
Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted
- **New Mutation Score:**
  \[
  \frac{\#\text{Killed} - \#\text{Killed Duplicates}}{\#\text{Mutants} - \#\text{Equivalent} - \#\text{Duplicates}}
  \]
- Detecting equivalent mutant 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?

Idea? 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?
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.
  - $\text{Cov_{stmt}(T_1)} > \text{Cov_{stmt}(T_2)} \rightarrow ?$
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  \[
  \text{Cov}_{\text{stmt}}(T_1) > \text{Cov}_{\text{stmt}}(T_2) \quad \rightarrow \quad T_1 \text{ is more likely to find more bugs?}
  \]
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $\text{Cov}_{\text{stmt}}(T_1) > \text{Cov}_{\text{stmt}}(T_2) \rightarrow \text{T1 is more likely to find more bugs?}$

What if you change $|T|$?
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $\text{Cov}_{\text{stmt}}(T_1) > \text{Cov}_{\text{stmt}}(T_2) \rightarrow$ T1 is more likely to find more bugs?
  - $\text{Cov}_{\text{stmt}}(T)$ increases with the $|T|$
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $\text{Cov}_{\text{stmt}}(T_1) > \text{Cov}_{\text{stmt}}(T_2) \rightarrow T_1$ is more likely to find more bugs?
  - $\text{Cov}_{\text{stmt}}(T)$ increases with the $|T|$
    $\rightarrow$ You cannot assume that better coverage increases defect finding ability!

Then does coverage serve a purpose?
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $\text{Cov}_{\text{stmt}}(T_1) > \text{Cov}_{\text{stmt}}(T_2) \rightarrow T_1 \text{ is more likely to find more bugs?}$
  - $\text{Cov}_{\text{stmt}}(T) \text{ increases with the } |T|$
    $\rightarrow \text{You cannot assume that better coverage increases defect finding ability!}$
  - Coverage still tells you which portions of a program haven't been tested!
Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - $\text{Cov}_{\text{stmt}}(T_1) > \text{Cov}_{\text{stmt}}(T_2) \rightarrow T1$ is more likely to find more bugs?
  - $\text{Cov}_{\text{stmt}}(T)$ increases with the $|T|$ → You cannot assume that better coverage increases defect finding ability!
  - Coverage still tells you which portions of a program haven't been tested!
  - It just cannot safely measure defect finding capability.
Traditional Coverage vs Mutation

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

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

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

```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.
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) {
    ...
}

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
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 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
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
Using Test Suites
For Other Purposes
Bug Prediction

- Does Bug Prediction Support Human Developers? Findings From a Google Case Study
Fault Localization

- **Doric: Foundations for Statistical Fault Localisation**

- **An Empirical Study of Fault Localization Families and Their Combinations**
Automated Program Repair

- Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts