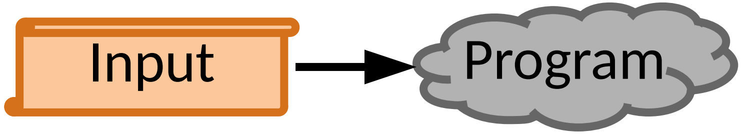
CMPT 745
Software Engineering

An Overview of Software Testing

Nick Sumner
wsumner@sfu.ca

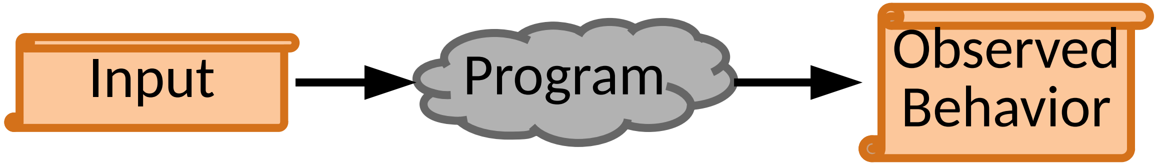
Software Testing

- The most common way of measuring and ensuring program correctness



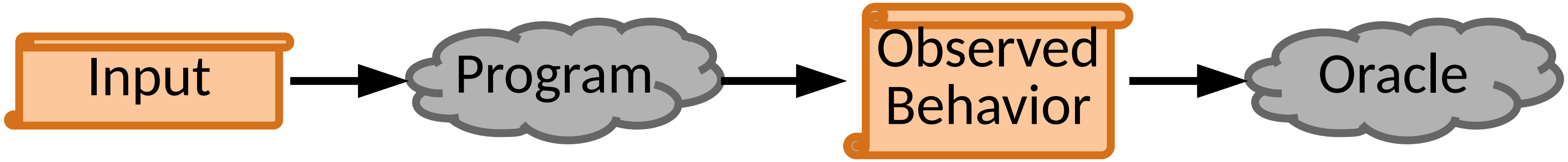
Software Testing

- The most common way of measuring and ensuring program correctness



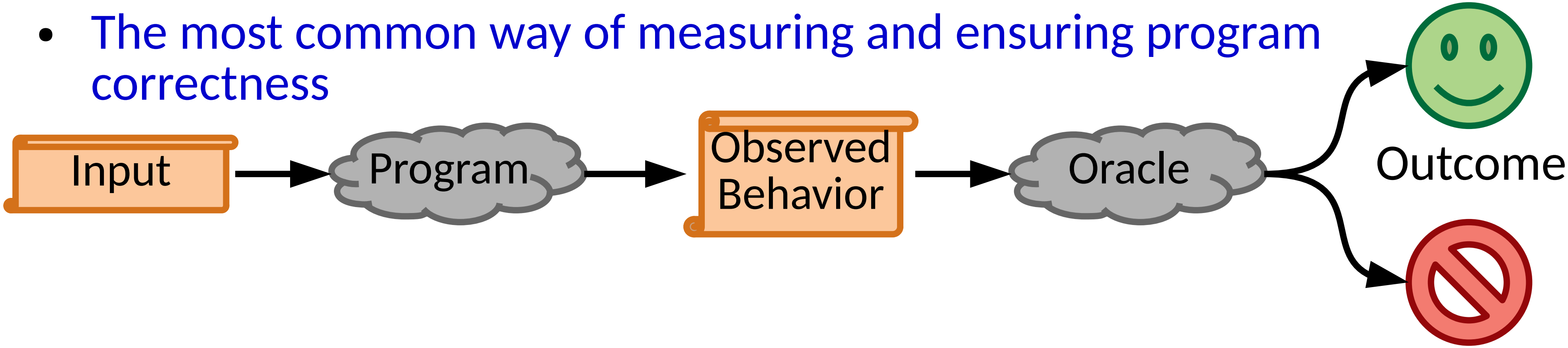
Software Testing

- The most common way of measuring and ensuring program correctness



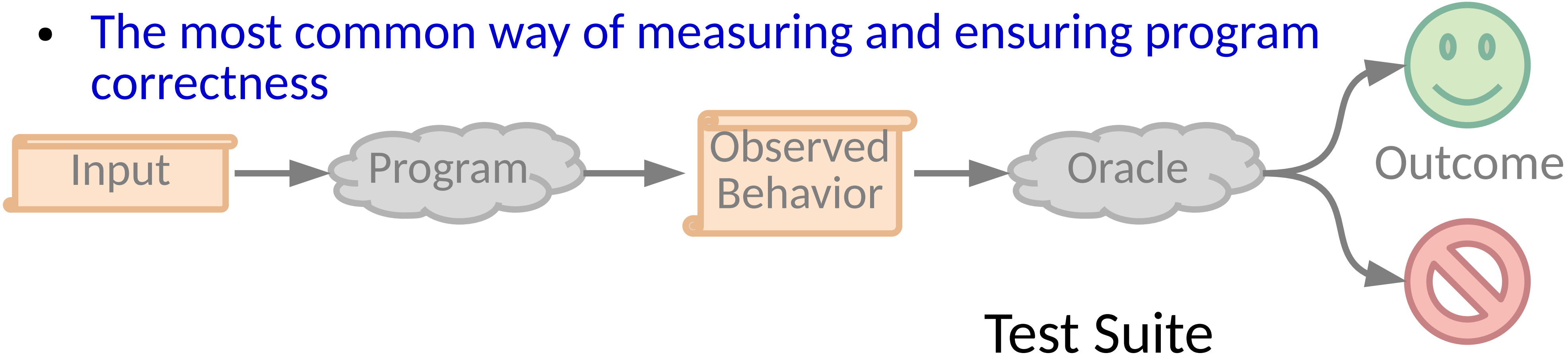
Software Testing

- The most common way of measuring and ensuring program correctness



Software Testing

- The most common way of measuring and ensuring program correctness

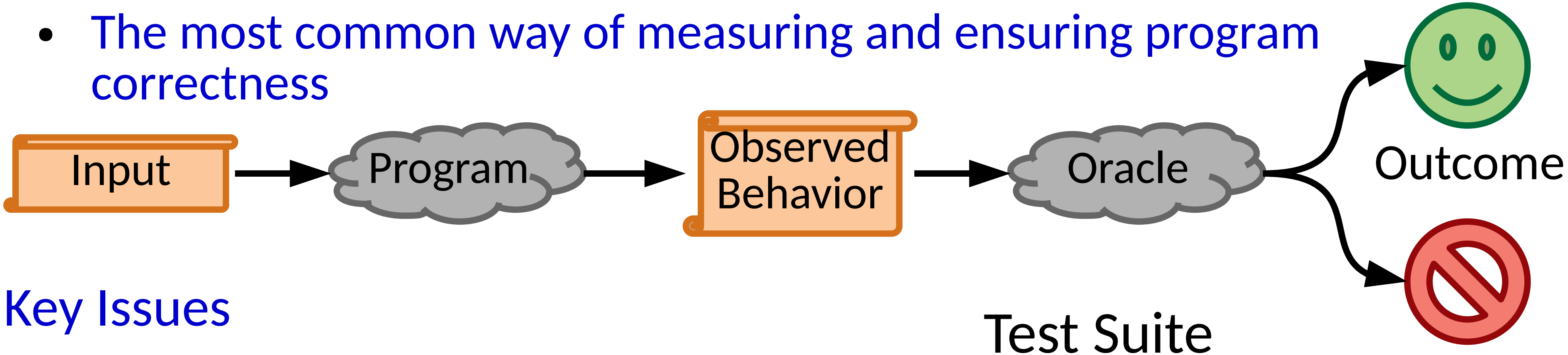


Test Suite

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

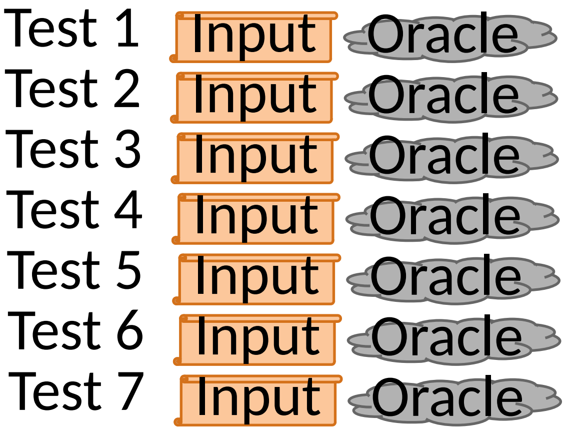
Software Testing

- The most common way of measuring and ensuring program correctness



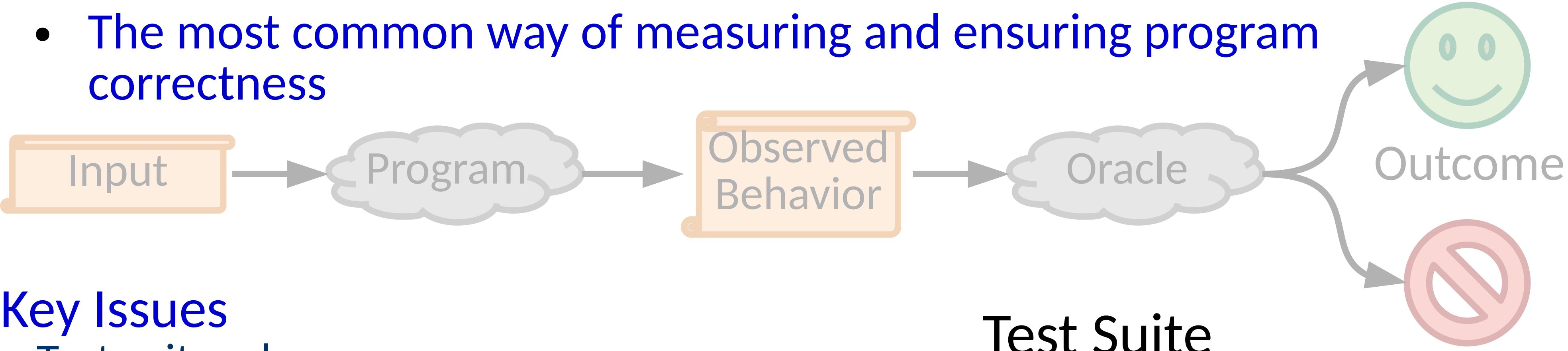
Key Issues

Test Suite



Software Testing

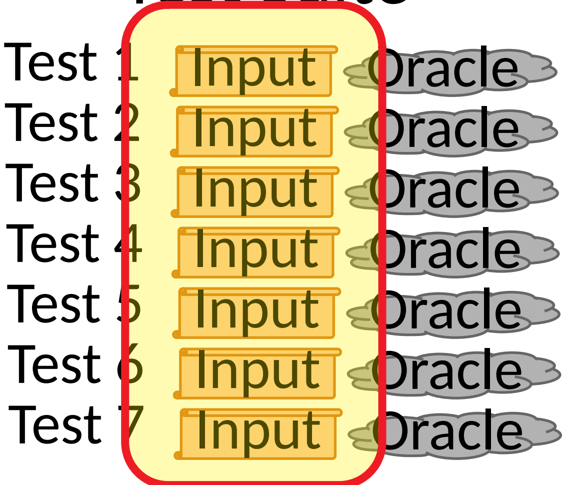
- The most common way of measuring and ensuring program correctness



Key Issues

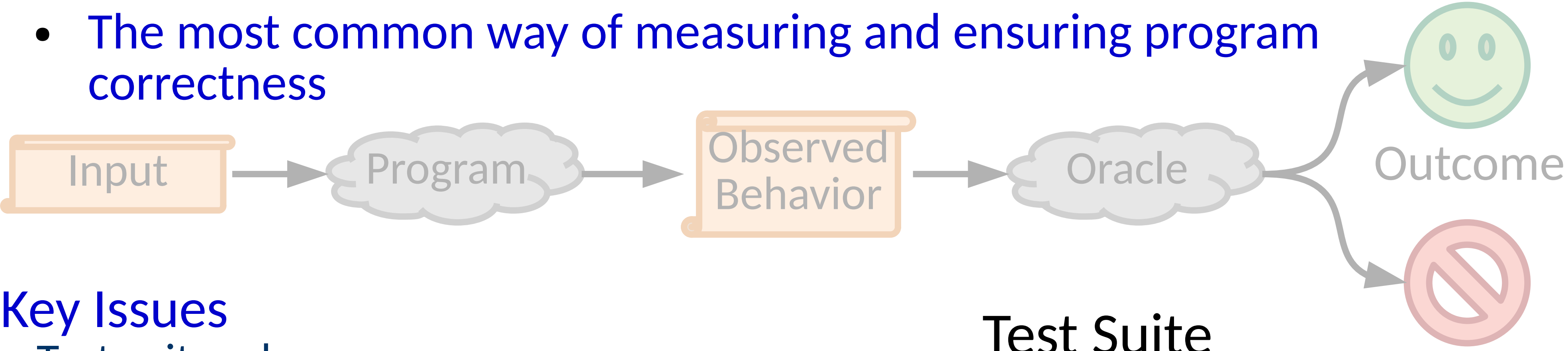
- Test suite adequacy

Test Suite



Software Testing

- The most common way of measuring and ensuring program correctness



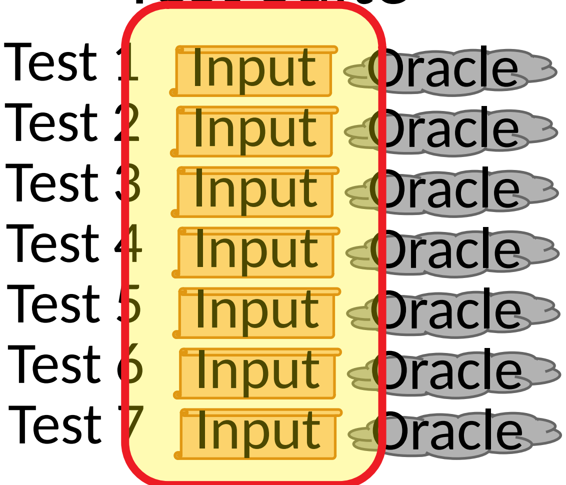
Key Issues

- Test suite adequacy

Testing is sampling.

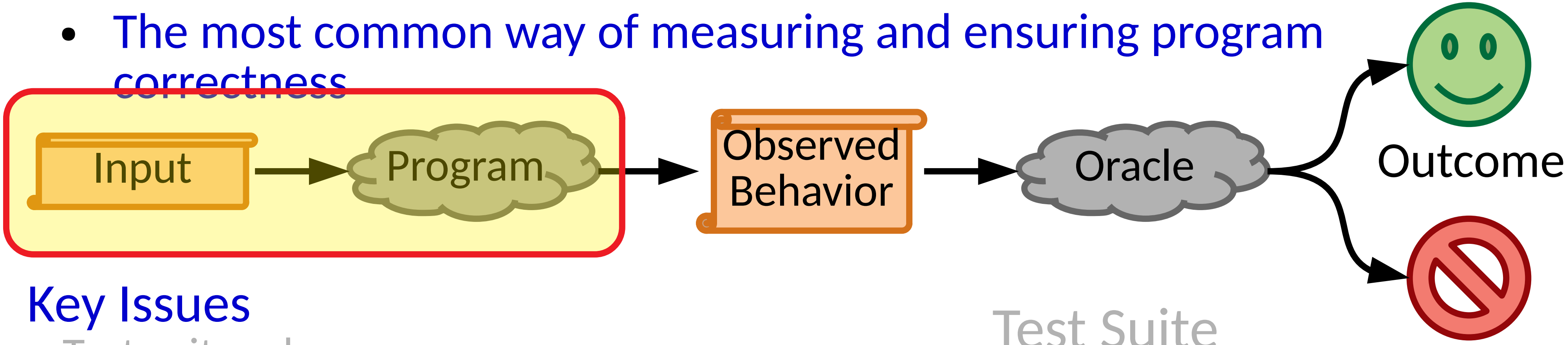
How do we know whether we are sampling well?

Test Suite



Software Testing

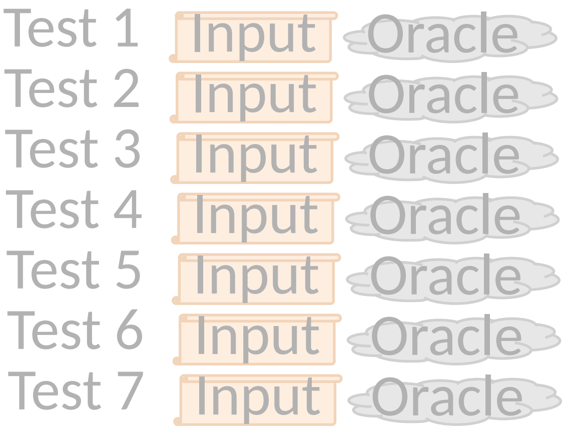
- The most common way of measuring and ensuring program correctness



Key Issues

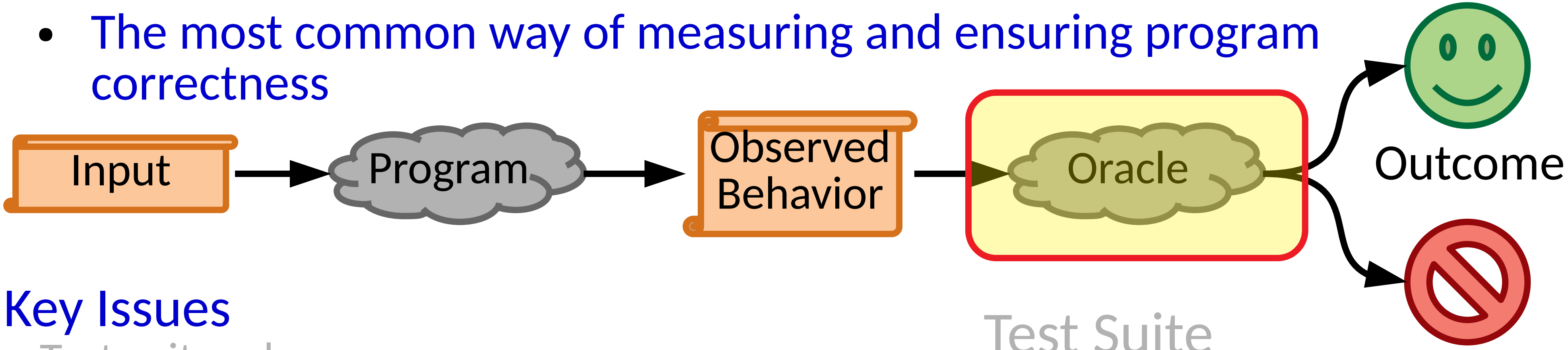
- Test suite adequacy
- Automated input generation

Test Suite



Software Testing

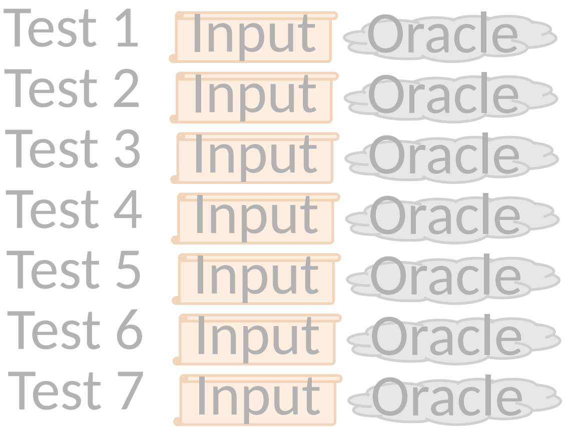
- The most common way of measuring and ensuring program correctness



Key Issues

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

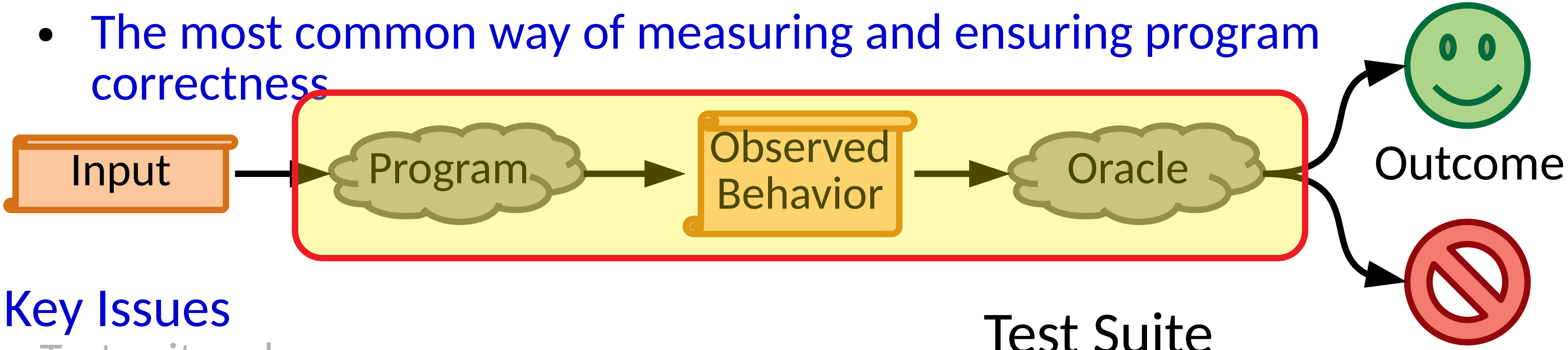
Test Suite



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

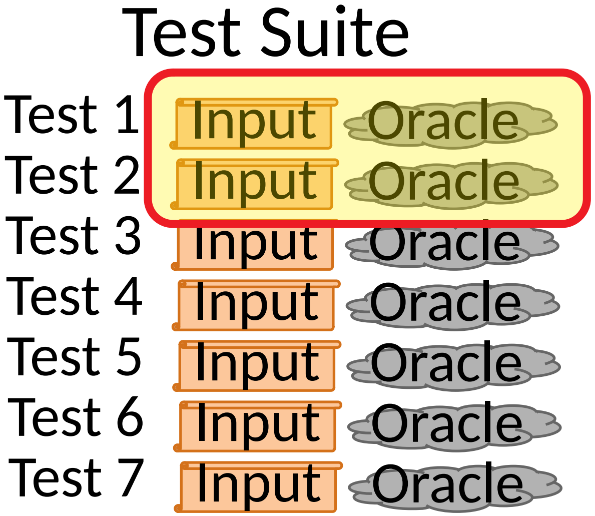
Software Testing

- The most common way of measuring and ensuring program correctness



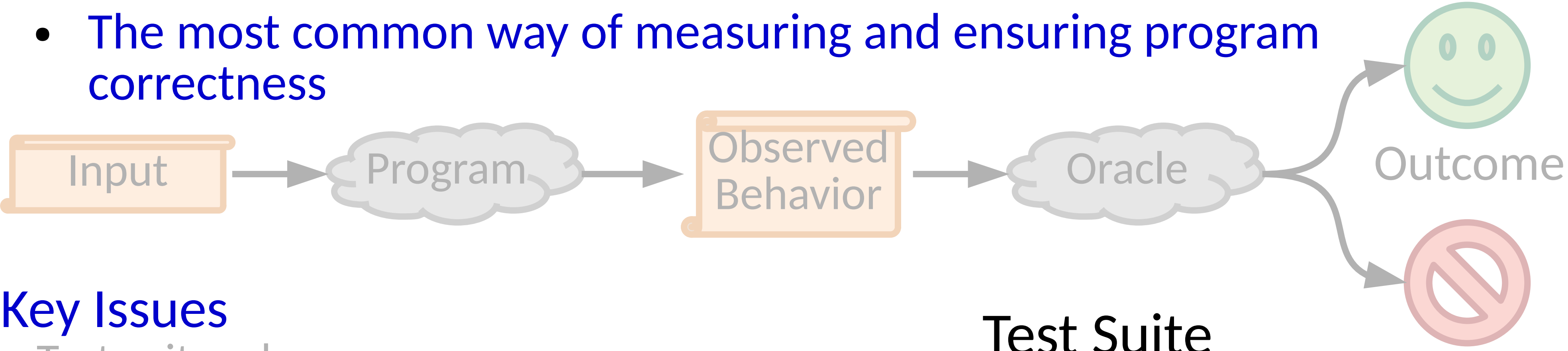
Key Issues

- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability



Software Testing

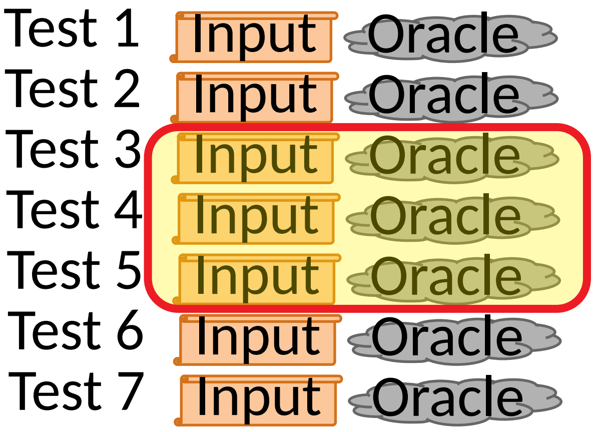
- The most common way of measuring and ensuring program correctness



Key Issues

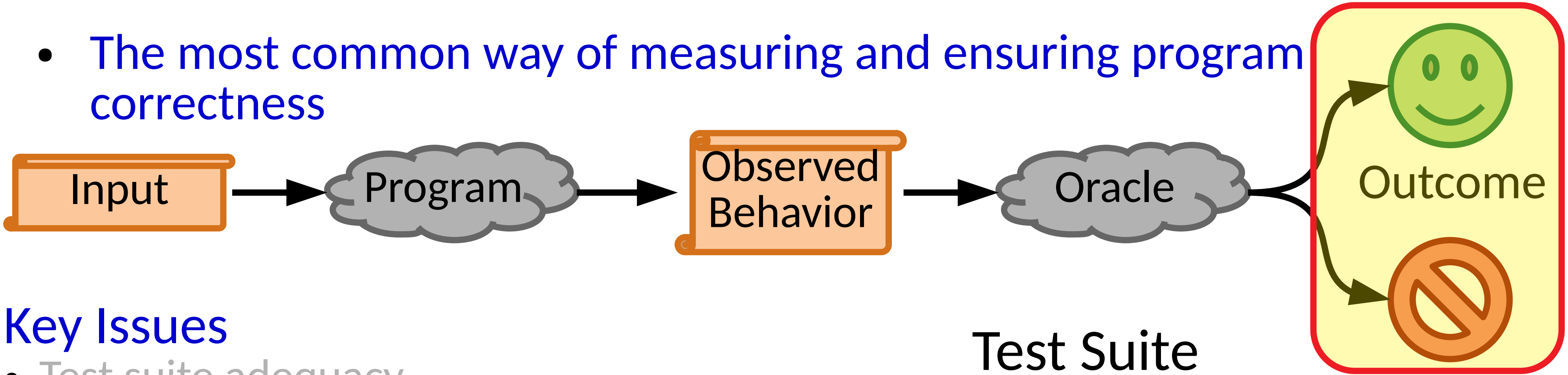
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
- Regression test selection

Test Suite



Software Testing

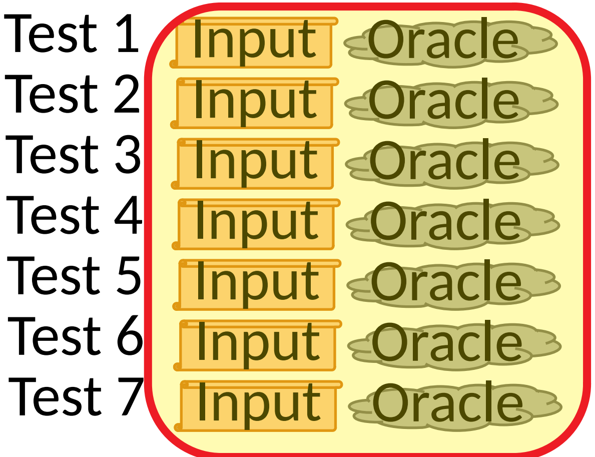
- The most common way of measuring and ensuring program correctness



Key Issues

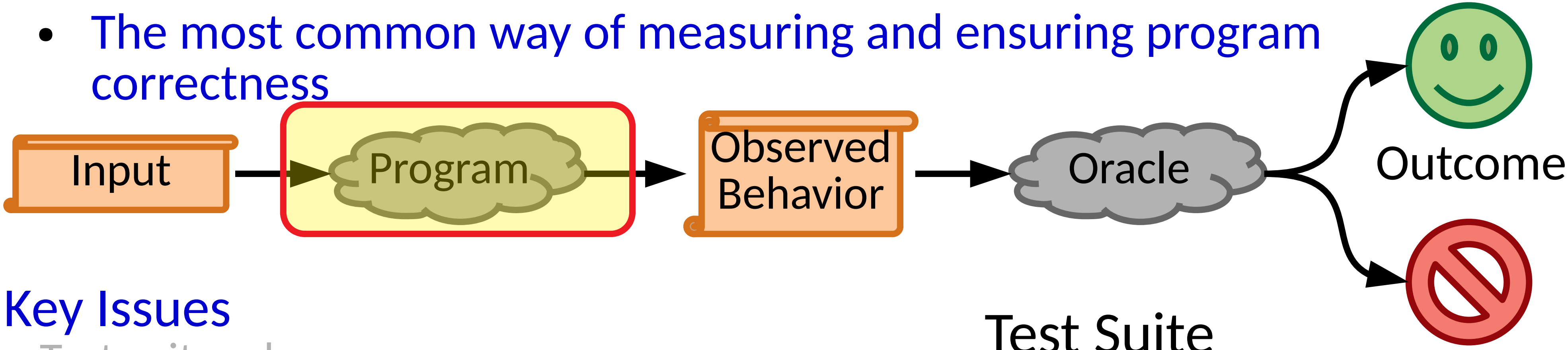
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
- Regression test selection
- Fault localization & automated debugging

Test Suite



Software Testing

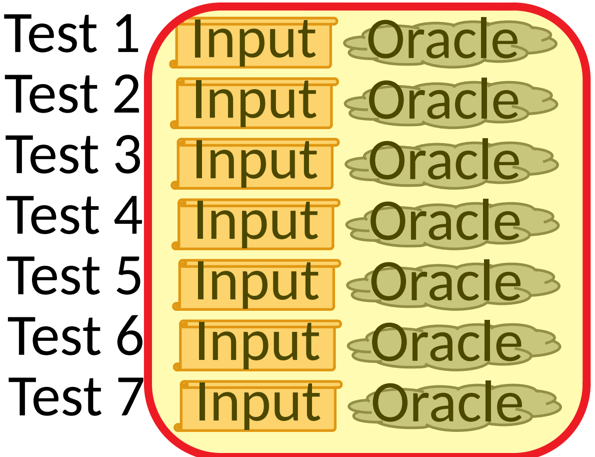
- The most common way of measuring and ensuring program correctness



Key Issues

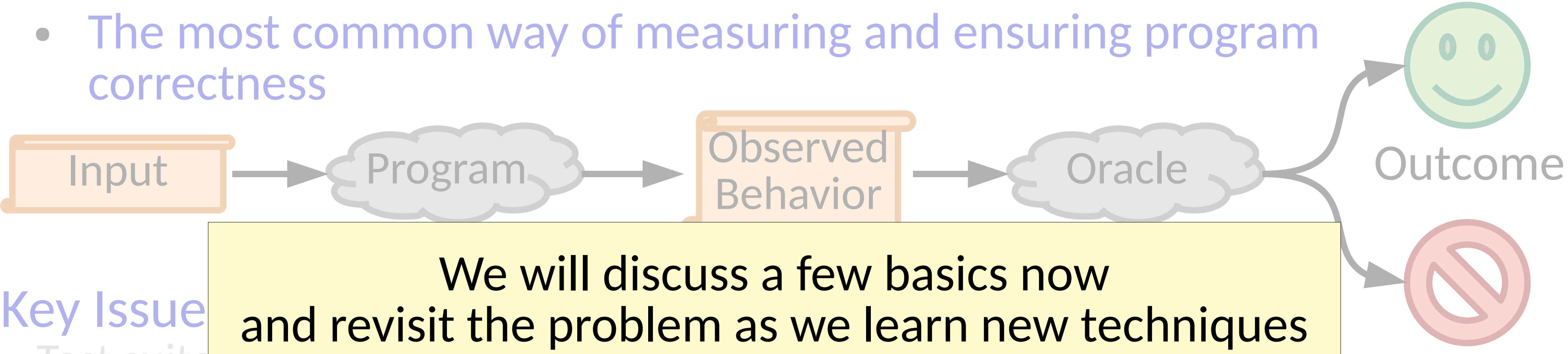
- Test suite adequacy
- Automated input generation
- Automated oracle generation
- Robustness/flakiness/maintainability
- Regression test selection
- Fault localization & automated debugging
- Automated program repair
- …

Test Suite



Software Testing

- The most common way of measuring and ensuring program correctness



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

Key Issue

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

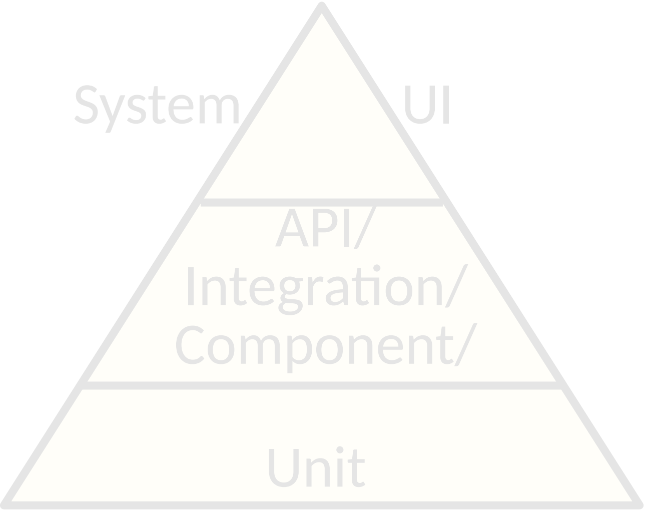
Test 1	Input	Oracle
Test 2	Input	Oracle
Test 3	Input	Oracle
Test 4	Input	Oracle
Test 5	Input	Oracle
Test 6	Input	Oracle
Test 7	Input	Oracle

Test Suite Design

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

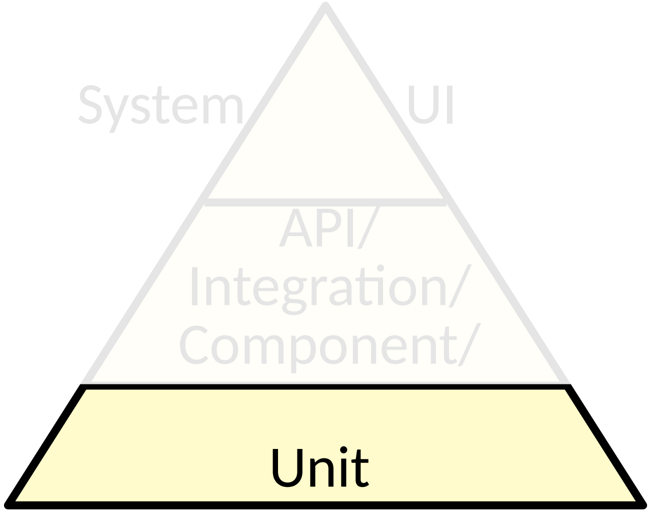
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)
- Components – The Automated Testing Pyramid



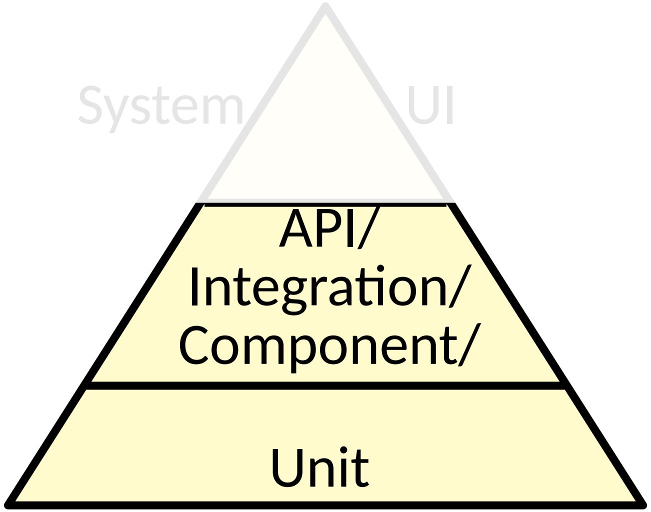
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)
- Components – The Automated Testing Pyramid



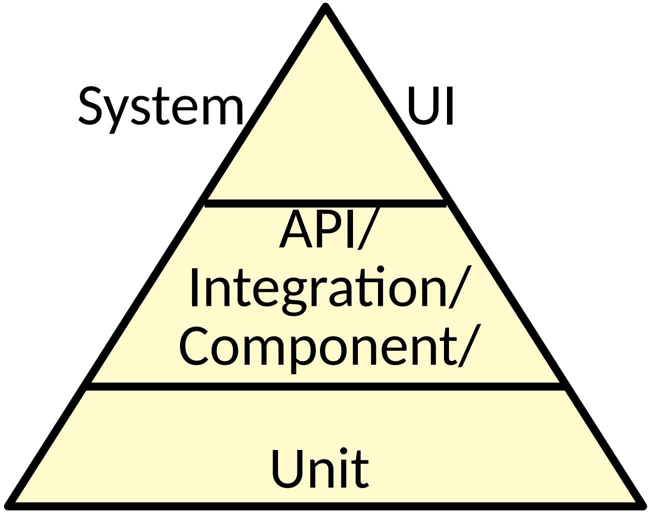
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, …)
- Components – The Automated Testing Pyramid



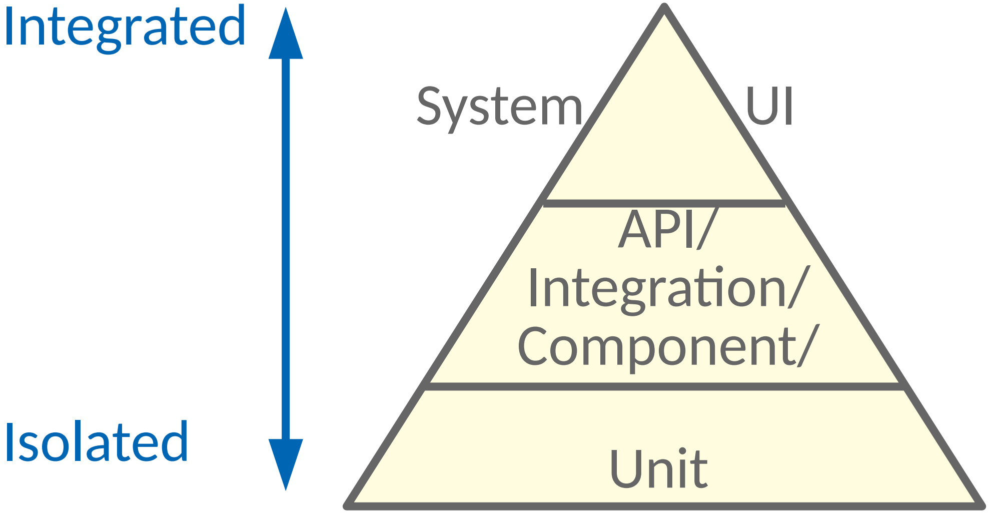
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, …)
- Components – The Automated Testing Pyramid



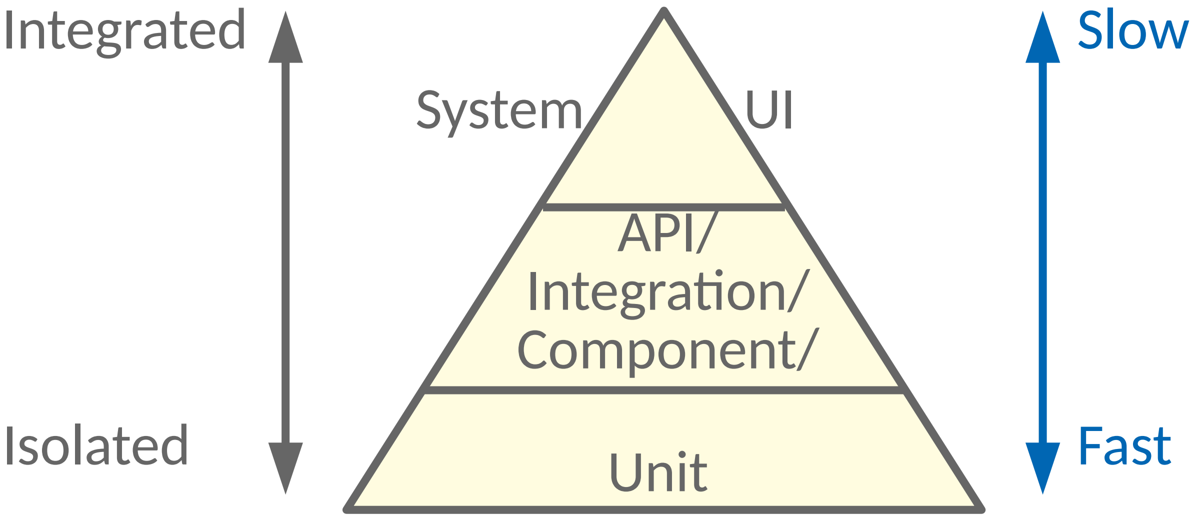
Test Suite Design

- 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

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

  auto result = evaluate(tree, env);

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

Designing a Unit Test

- Common structure

```
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

```
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

```
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

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

Designing a Unit Test

- Common structure
- Tests should run in isolation

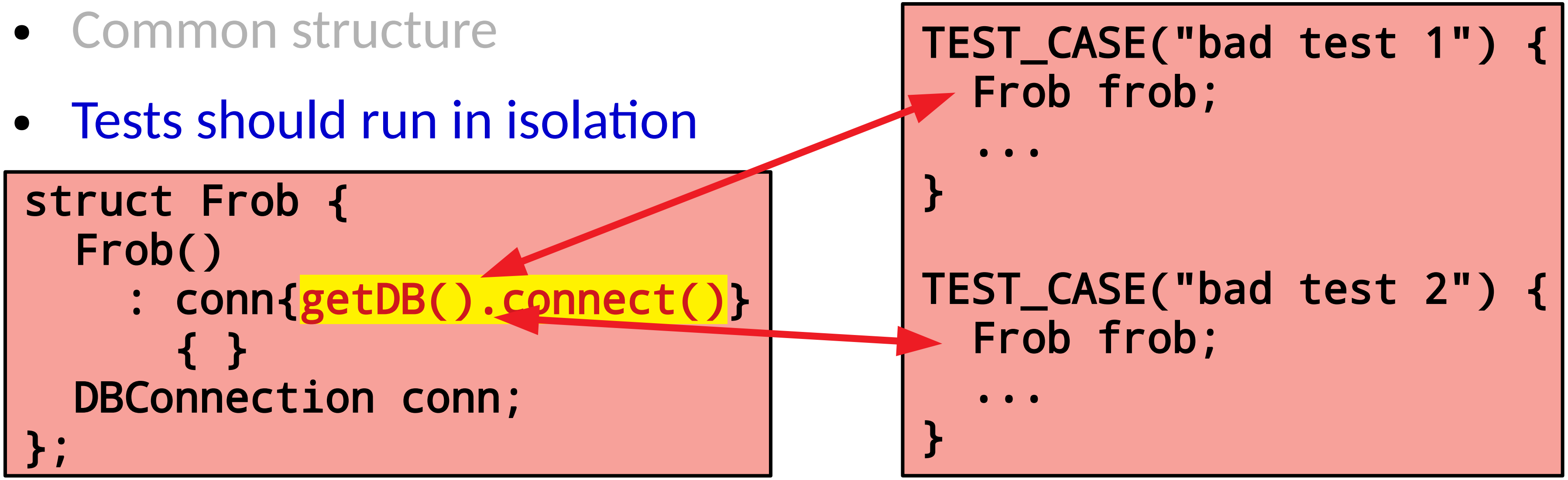
```
struct Frob {
  Frob()
    : conn{getDB().connect()}
      { }
  DBConnection conn;
};
```

```
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



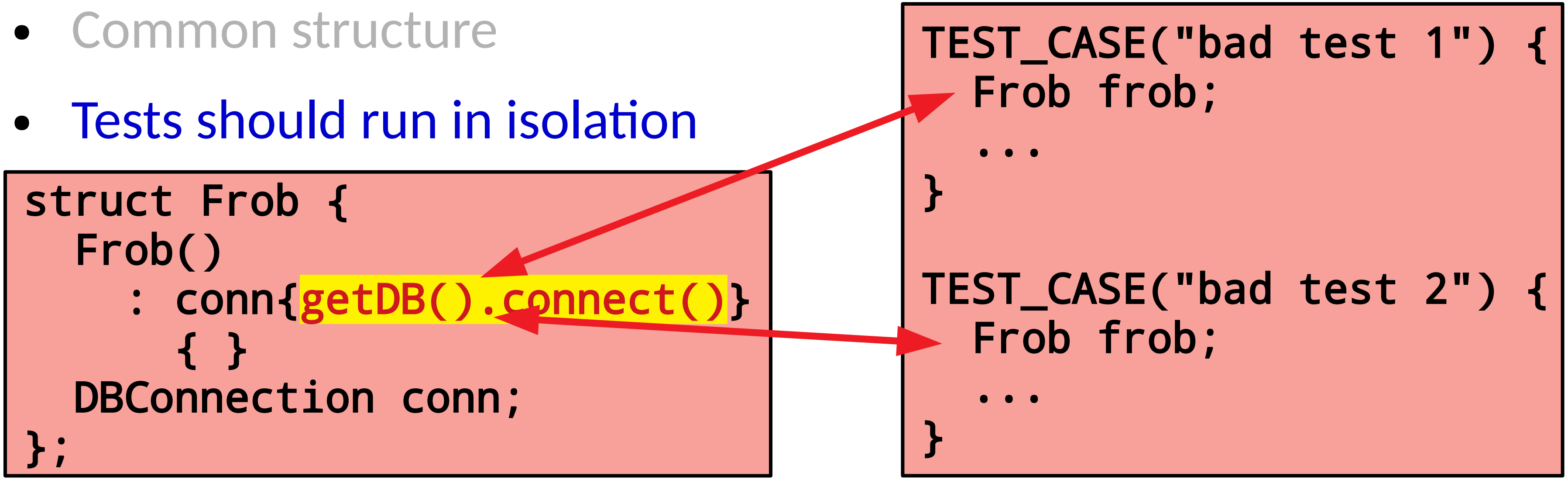
```
struct Frob {
  Frob()
    : conn{getDB().connect()}
      { }
  DBConnection conn;
};
```

```
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



```
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

```
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!

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

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

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

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

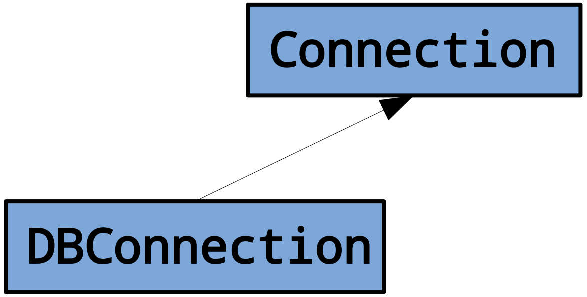
Connection

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;
};
```



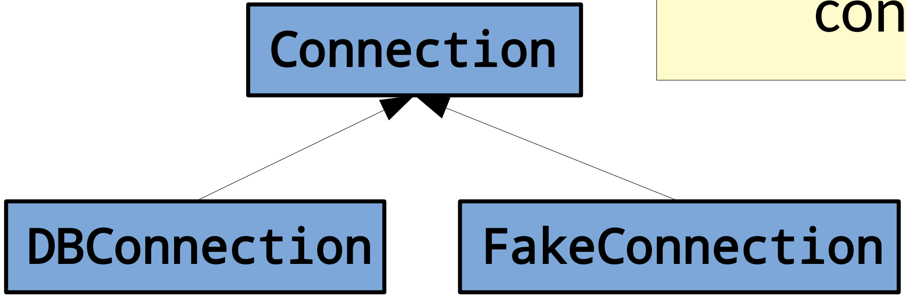
Dependency injection allows the user of a class to control its behavior

Designing a Unit Test

- Common structure
- Tests should run in isolation

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

Dependency injection allows the user of a class to control its behavior

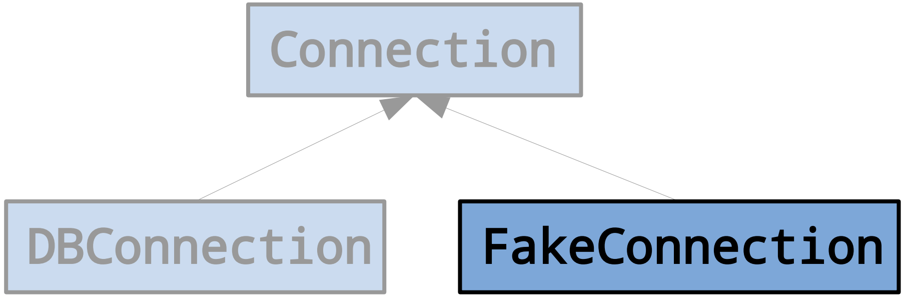


Designing a Unit Test

- Common structure
- Tests should run in isolation

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

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

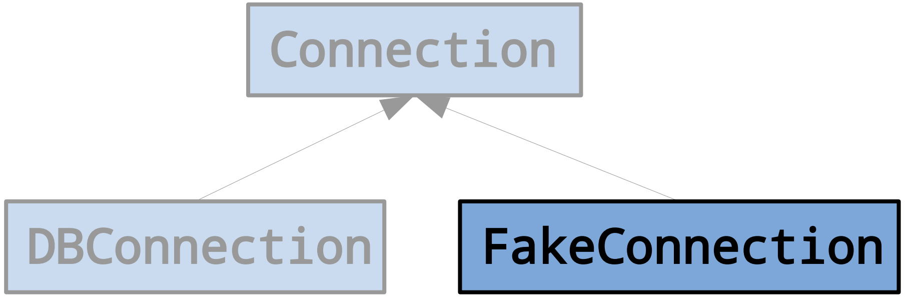


Designing a Unit Test

- Common structure
- Tests should run in isolation

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

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



Connection

DBConnection

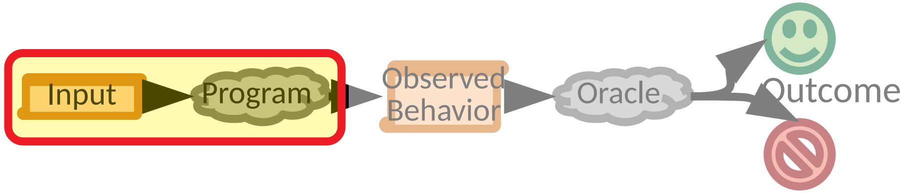
FakeConnection

Mocks & stubs isolate and examine how a component interacts with dependencies

Designing a Unit Test

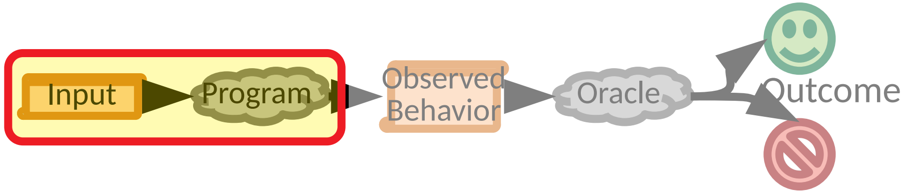
- Common structure
- Tests should run in isolation
- Key problem to resolve:
  - How do you define your inputs & oracles?

Selecting Inputs



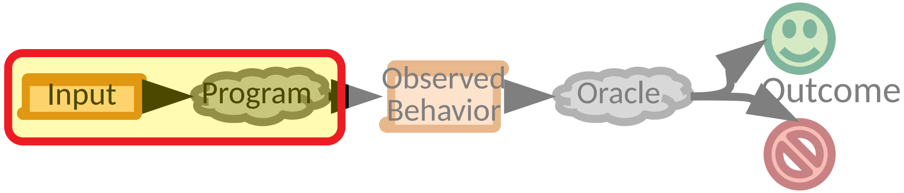
- Two broad categories

Selecting Inputs



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

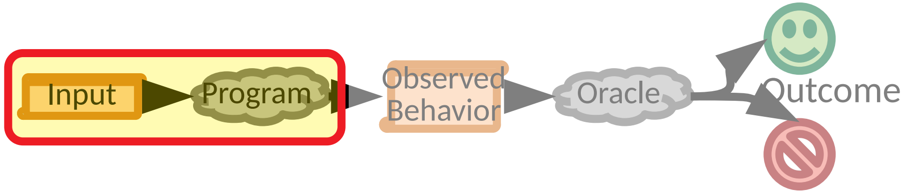
Selecting Inputs



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

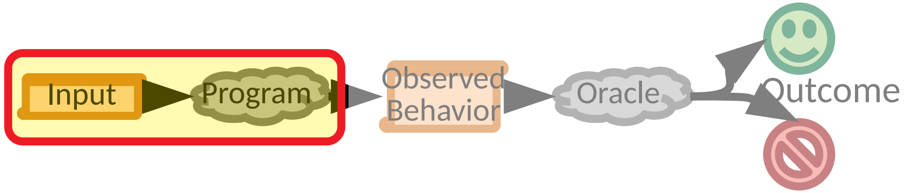
specification based (BDD?)
model driven
naive fuzzing
boundary value analysis

Selecting Inputs



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

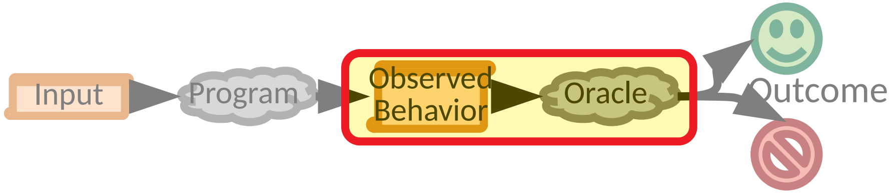
Selecting Inputs



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

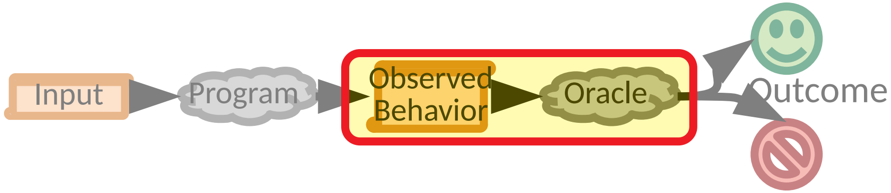
symbolic execution
call chain synthesis
grey/whitebox fuzzing

Designing Oracles



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

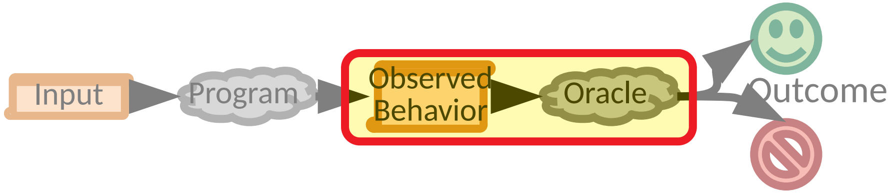
Designing Oracles



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

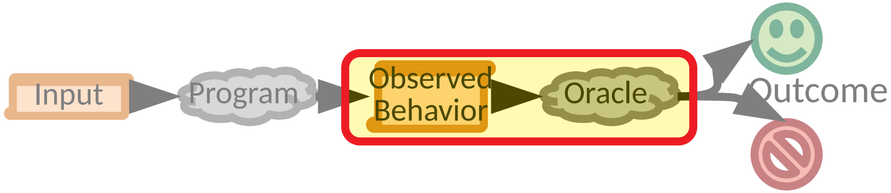
What about tasks like:
machine learning
simulation
…

Designing Oracles



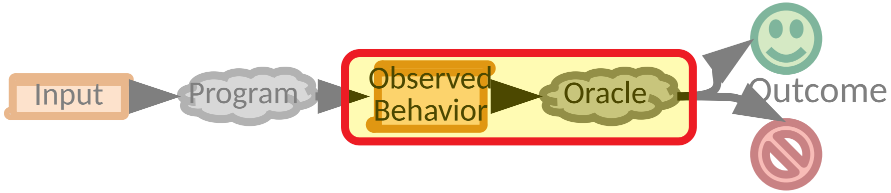
- Sometimes it is simple
  - For a known scenario, a specific output is expected
- Invariants & properties are powerful

Designing Oracles



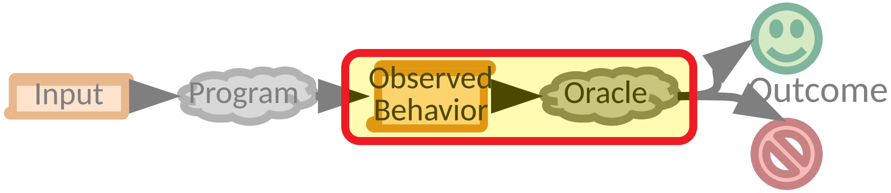
- Sometimes it is simple
  - For a known scenario, a specific output is expected
- Invariants & properties are powerful
  - $foo^{-1}(foo(x)) == x$ (e.g. archive & unarchive a file)

Designing Oracles



- Sometimes it is simple
  - For a known scenario, a specific output is expected
- Invariants & properties are powerful
  - $foo^{-1}(foo(x))$ == x (e.g. archive & unarchive a file)
  - turn(360, direction) == direction

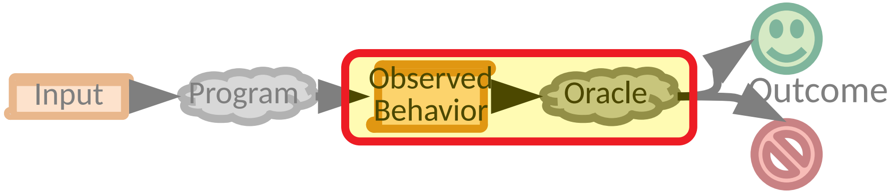
Designing Oracles



- 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 & un
  - turn(360, direction) == direction

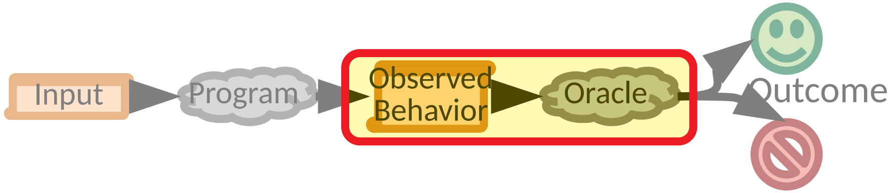
Metamorphic testing

Designing Oracles



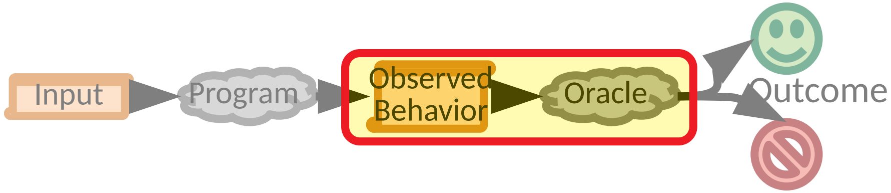
- Sometimes it is simple
  - For a known scenario, a specific output is expected
- Invariants & properties are powerful
  - foo^{-1}(foo(x)) == x (e.g. archive & unarchive a file)
  - turn(360, direction) == direction
  - program1(x) == program2(x)

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)

Differential testing

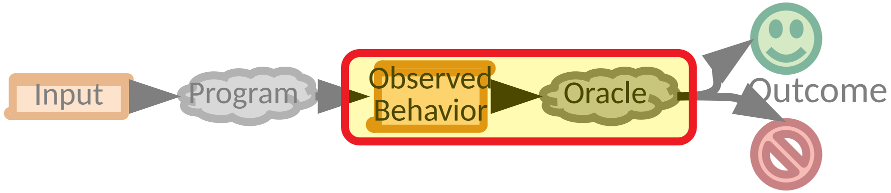


Designing Oracles

- 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)

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?

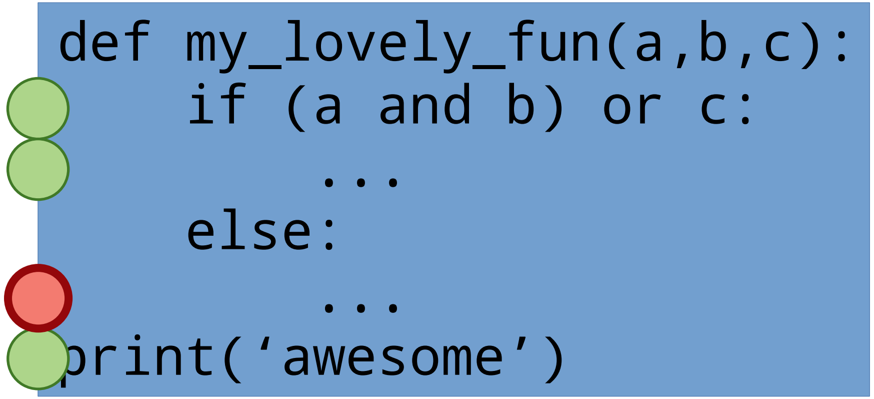
Test Suite Adequacy

- Metrics

Remember: A higher score ***should*** mean fewer defects

Test Suite Adequacy

- Metrics
  - Statement coverage



```
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?

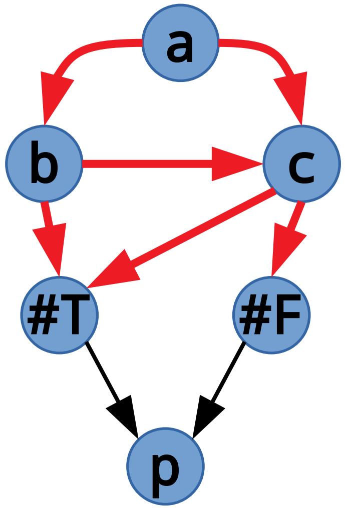
$$\text{score} = \frac{\#\ \text{covered}}{\#\ \text{statements}}$$

Test Suite Adequacy

- Metrics
  - Statement coverage
  - Branch coverage

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

We will discuss
control flow graphs
again soon

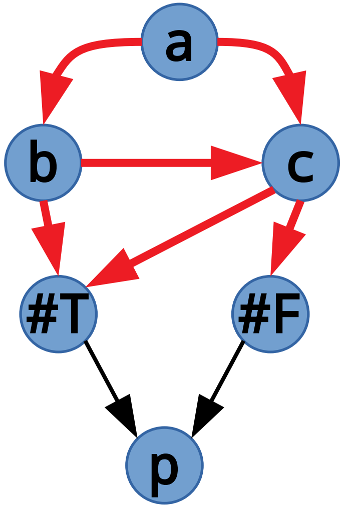


$$\text{score} = \frac{\#\ \text{covered}}{\#\ \text{branches}}$$

Test Suite Adequacy

- Metrics
  - Statement coverage
  - Branch coverage

```
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?

$$\text{score} = \frac{\text{\# covered}}{\text{\# branches}}$$

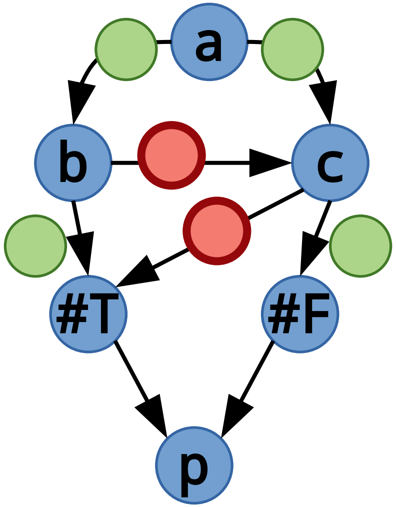
Test Suite Adequacy

- Metrics
  - Statement coverage
  - Branch coverage

```
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

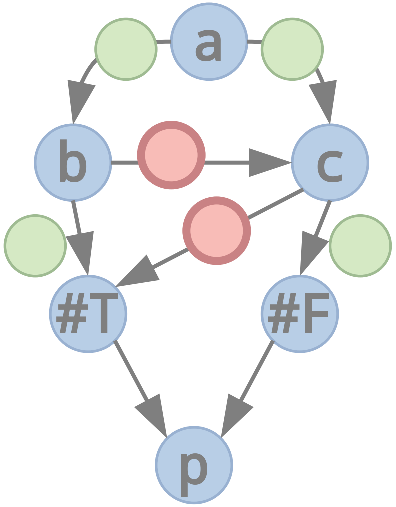
```
def my_lovely_fun(a,b,c):
    if (a and b) or c:
        ...
    else:
        ...
print('awesome')
```

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?

```
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?

```
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?

```
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?

```
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

a in this condition
is covered by the test suite

Test Suite Adequacy

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

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

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

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')
```

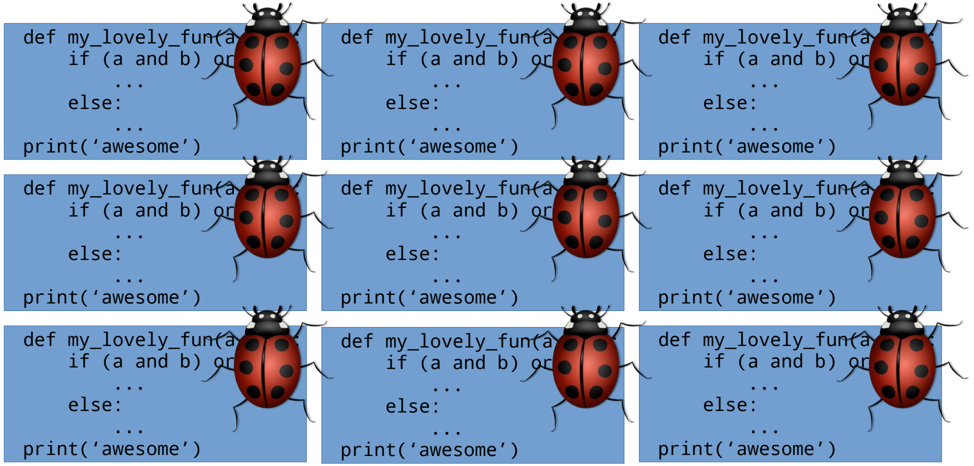
How many ***injected bugs***
can be detected by the test suite?

Test Suite Adequacy

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

```
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?



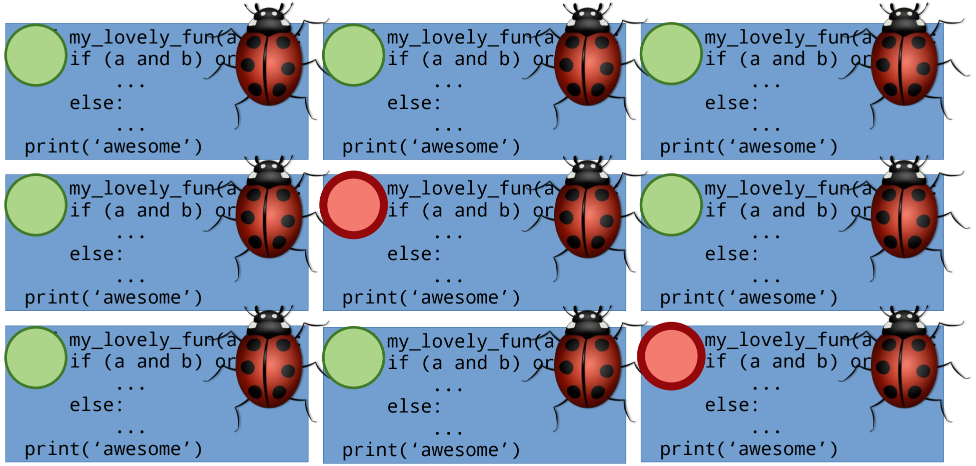
```
def my_lovely_fun(a
    if (a and b) or
        ...
    else:
        ...
print(‘awesome’)
```

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?



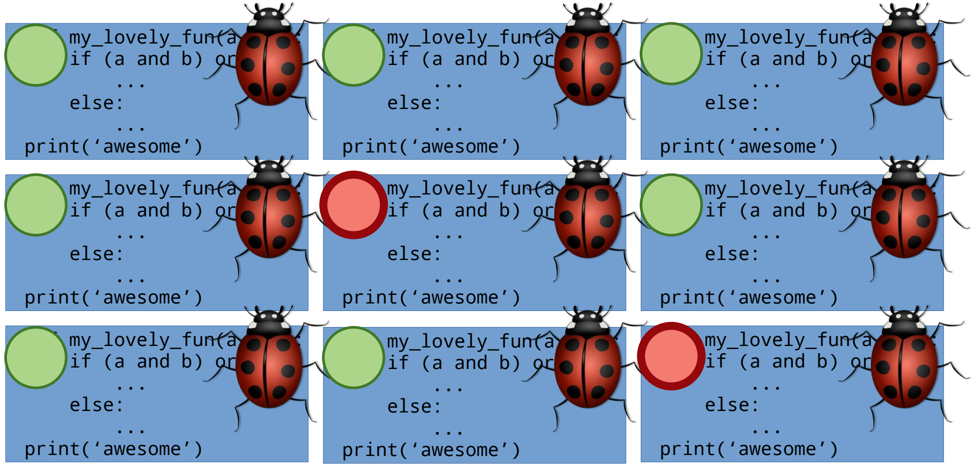
Test Suite Adequacy

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

```
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?

$$\text{score} = \frac{\text{\# covered/killed}}{\text{\# non-equivalent mutants}}$$

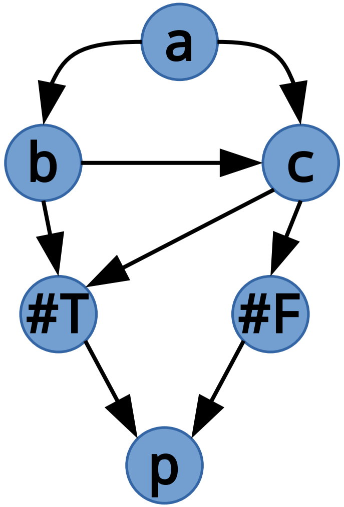


Test Suite Adequacy

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

```
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?



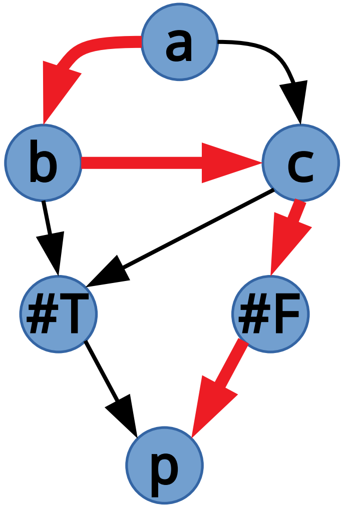
Test Suite Adequacy

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

```
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?

abT
abcT
abcF
acT
acF



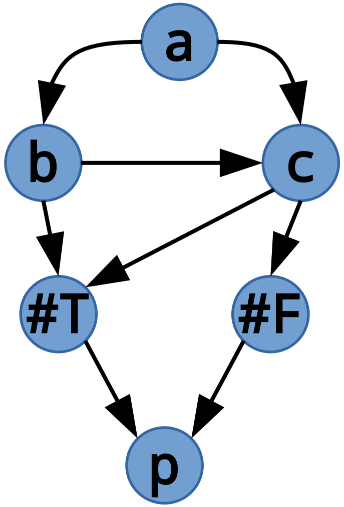
Test Suite Adequacy

- 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’)
```

🟢 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.

[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.

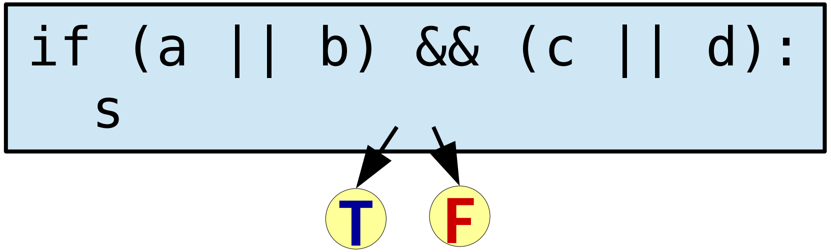
```
if (a || b) && (c || d):
  s
```

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

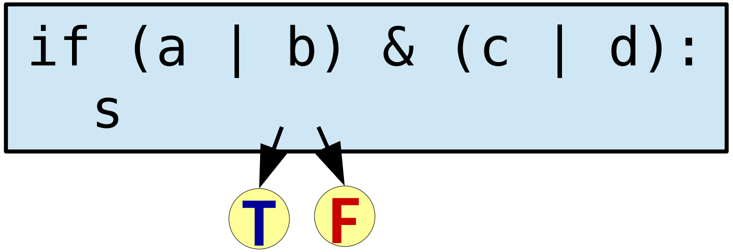
How does it do in these cases?

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
```

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.

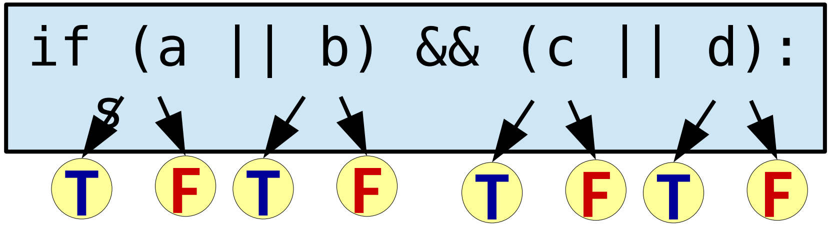
```
if (a || b) && (c || d):
  s
```

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

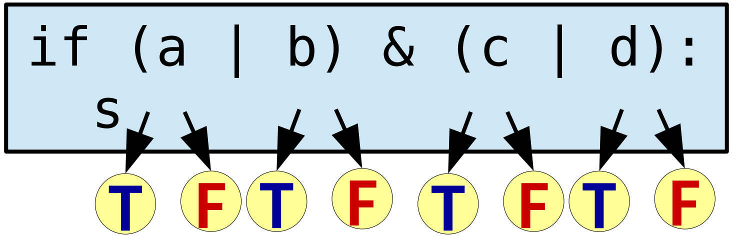
How does it do in these cases?

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.



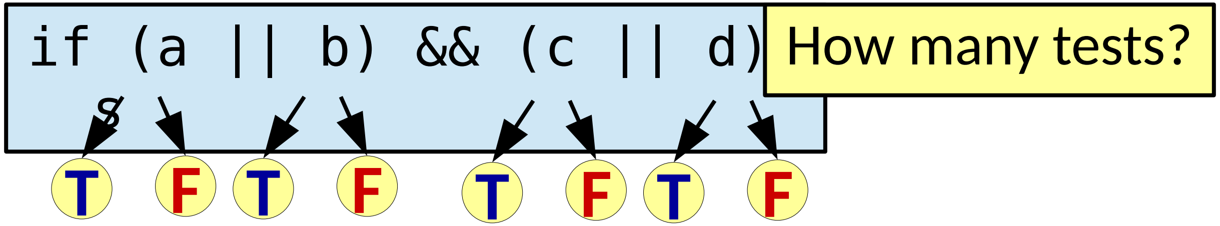
```
if (a || b) && (c || d):
   s
```



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

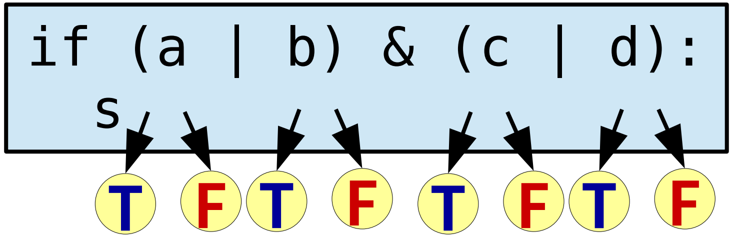
MC/DC Coverage

- 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
```

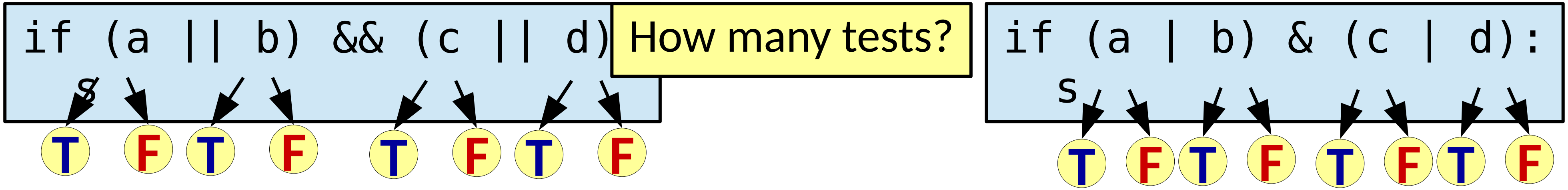
How many tests?



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

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
```

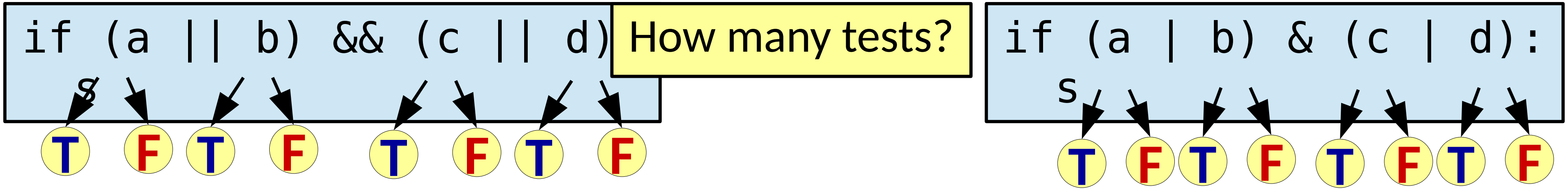
How many tests?

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

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
```

How many tests?

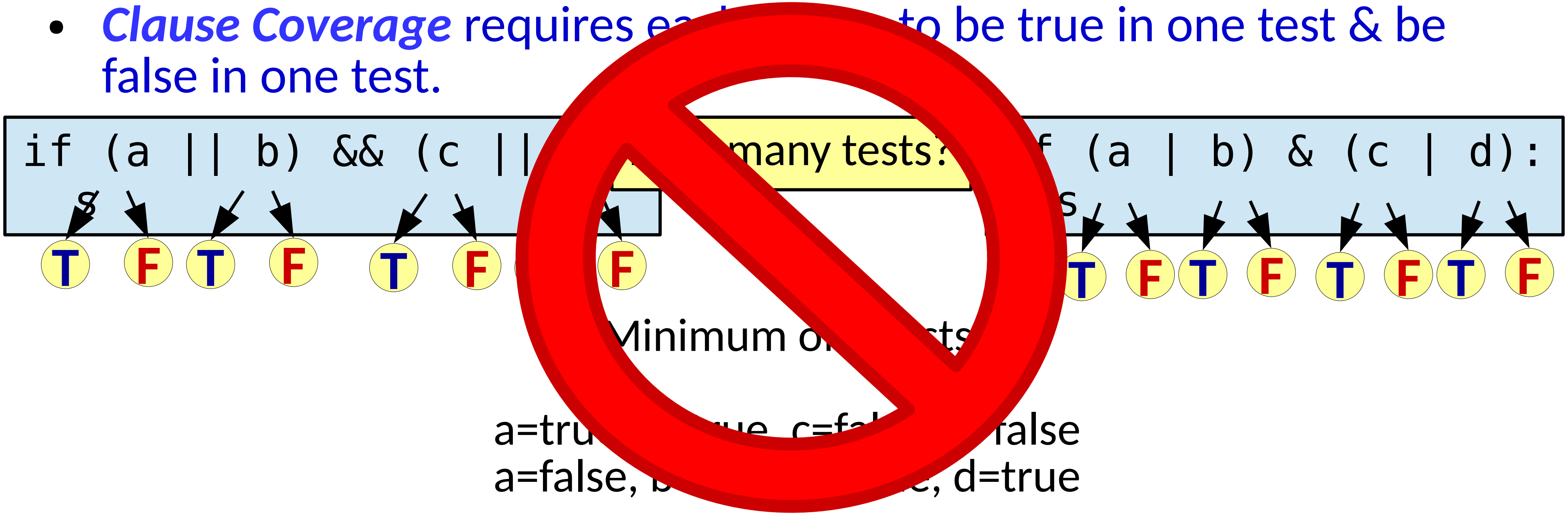
```
if (a | b) & (c | d):
  s
```

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 ea to be true in one test & be false in one test.



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

many tests?

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

Minimum o ts

a=tru ue, c=fal false
a=false, b e, 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

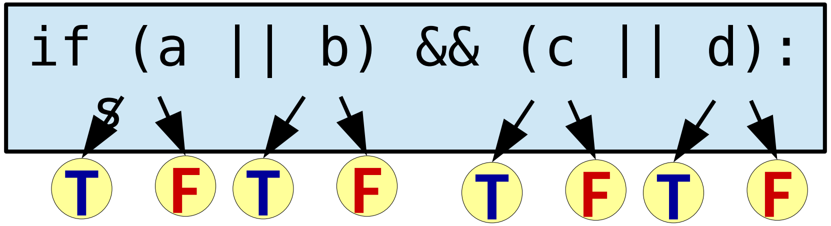
MC/DC Coverage

- *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



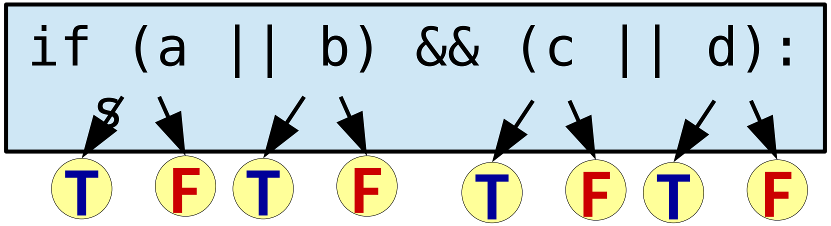
```
if (a || b) && (c || d):
```

Minimum of **2** tests

a=true, b=true, c=true, d=true
a=false, b=false, c=false, d=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 || b) && (c || d):
```

Is this good?

Minimum of **2** tests

a=true, b=true, c=true, d=true
a=false, b=false, c=false, d=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
  4) Each clause independently impacts the 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 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 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

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

```
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

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 || 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 || b && c)
```

a=F

b=T

c=T

$$\underline{}$$

T

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 || b && c)
```

a=F	a=F
b=T	b=T
c=T	c=F
T	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

$$\varphi(a,b,c) \neq \varphi(a,b,\neg c)$$

```
(a || b && c)
```

a=F	a=F
b=T	b=T
c=T	c=F
T	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

$$\varphi(a,b,c) \neq \varphi(a,b,\neg c)$$

```
(a || b && c)
```

a=F	a=F
b=T	b=T
c=T	c=F
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=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 |

a & b
If a=True, b determines
the outcome.

a | b
If a=False, 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 |

a & b	a | b
If a=True, b determines the outcome.	If a=False, b determines the outcome.

- By definition, solve $\varphi_{c=true} \oplus \varphi_{c=false}$

MC/DC Coverage

- Given **a | (b & c)**, generate tests for a

a has impact ↔

MC/DC Coverage

- Given **a | (b & c)**, generate tests for a

a has impact ↔ #T | (b & c) ≠ #F | (b & c)

MC/DC Coverage

- Given **a | (b & c)**, generate tests for a

a has impact ↔ #T | (b & c) ≠ #F | (b & c)

↔ #T ≠ b & c

MC/DC Coverage

- Given **a | (b & c)**, generate tests for a

a has impact ↔ #T | (b & c) ≠ #F | (b & c)

↔ #T ≠ b & c

↔ #T = ¬b | ¬c

MC/DC Coverage

- Given **a | (b & c)**, generate tests for a

a has impact ↔ #T | (b & c) ≠ #F | (b & c)

↔ #T ≠ b & c

↔ #T = ¬b | ¬c

↔ b is false or c is false

MC/DC Coverage

- Given **a | (b & c)**, generate tests for a

a has impact ↔ #T | (b & c) ≠ #F | (b & c)

↔ #T ≠ b & c

↔ #T = ¬b | ¬c

↔ b is false or c is false

defines two different ways to test a

MC/DC Coverage

- Given **a | (b & c)**, generate tests for a

a has impact ↔ #T | (b & c) ≠ #F | (b & c)

↔ #T ≠ b & c

↔ #T = ¬b | ¬c

↔ 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 | (b & c)**, generate tests for a

a has impact ↔ #T | (b & c) ≠ #F | (b & c)

↔ #T ≠ b & c

↔ #T = ¬b | ¬c

↔ 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

Have c be #F

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

MC/DC Coverage

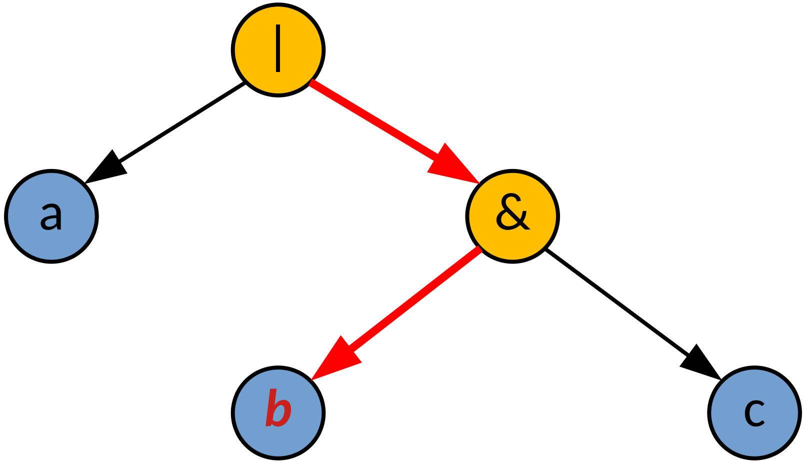
- Given **a | (b & c)**, generate tests for ***b***

  b has impact

MC/DC Coverage

- Given **a | (b & c)**, generate tests for ***b***

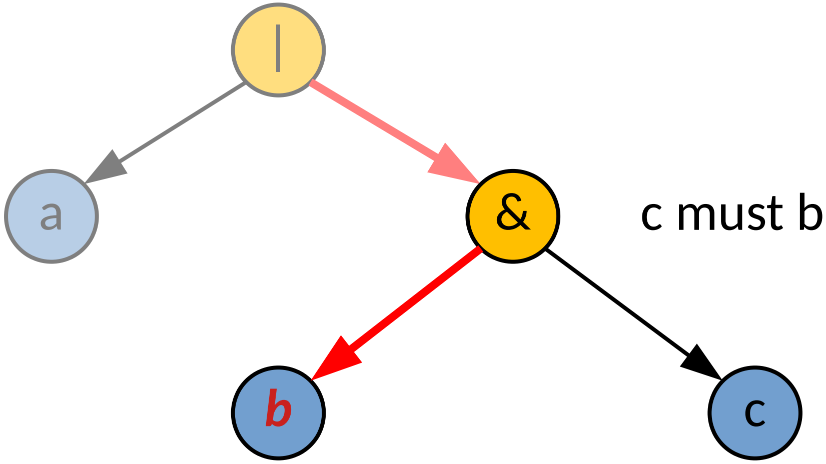
    b has impact



MC/DC Coverage

- Given **a | (b & c)**, generate tests for ***b***

b has impact



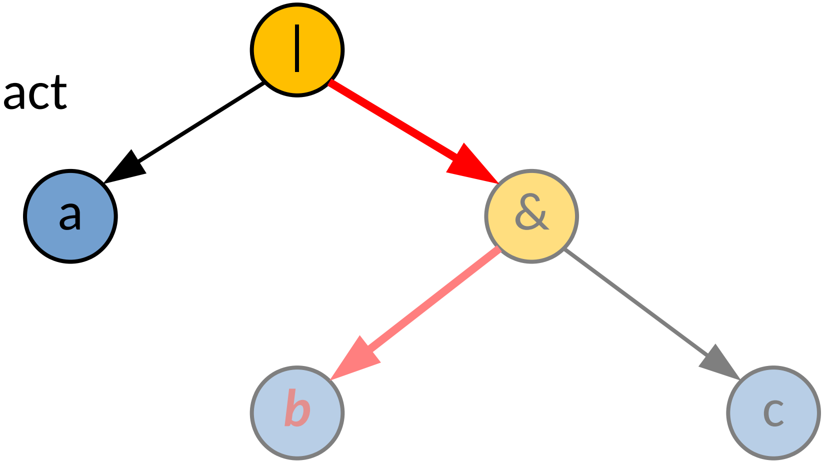
c must be true for impact

MC/DC Coverage

- Given **a | (b & c)**, generate tests for ***b***

b has impact

a must be false for impact



MC/DC Coverage

- Given **a | (b & c)**, generate tests for ***b***

b has impact ↔ a = #F & c = #T

MC/DC Coverage

- What about (a & b) | (a & ¬b)?
  - Can you show the impact of a?

MC/DC Coverage

- What about (a & b) | (a & ¬b)?
  - Can you show the impact of a?
  - Can you show the impact of b?

MC/DC Coverage

- What about (a & b) | (a & ¬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) | (a & ¬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 & b) | (a & ¬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) | (a & ¬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) | (a & ¬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

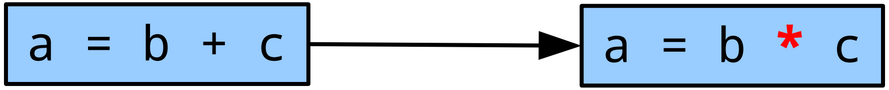
Mutation Analysis

- 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



```
a = b + c
```

```
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?

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

Mutation

- What are possible mutants?

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

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

Mutation

- What are possible mutants?

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

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

- Some are not generally useful:
  - (*Still Born*) Not compilable

Mutation

- What are possible mutants?

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

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

- Some are not generally useful:
  - (*Still Born*) Not compilable
  - (*Trivial*) Killed by most test cases

Mutation

- What are possible mutants?

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

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

- Some are not generally useful:
  - (*Still Born*) Not compilable
  - (*Trivial*) Killed by most test cases
  - (*Equivalent*) Indistinguishable from original program

Mutation

- What are possible mutants?

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

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

```
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

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

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

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

  if (b < a) {


    minVal = b;



  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

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


    minVal = b;




  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

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

    minVal = b;




  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

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



        }
        return minVal;
      }
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

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


  }
  return minVal;
}
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

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

              }
              return minVal;
            }
```

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

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

- Mimic mistakes
- Encode knowledge from other techniques

Mutation

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

What mimics statement coverage?

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

- Mimic mistakes
- Encode knowledge from other techniques

Mutation Analysis

Mutants

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

Mutation Analysis

Mutants

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

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

Mutation Analysis

Mutants

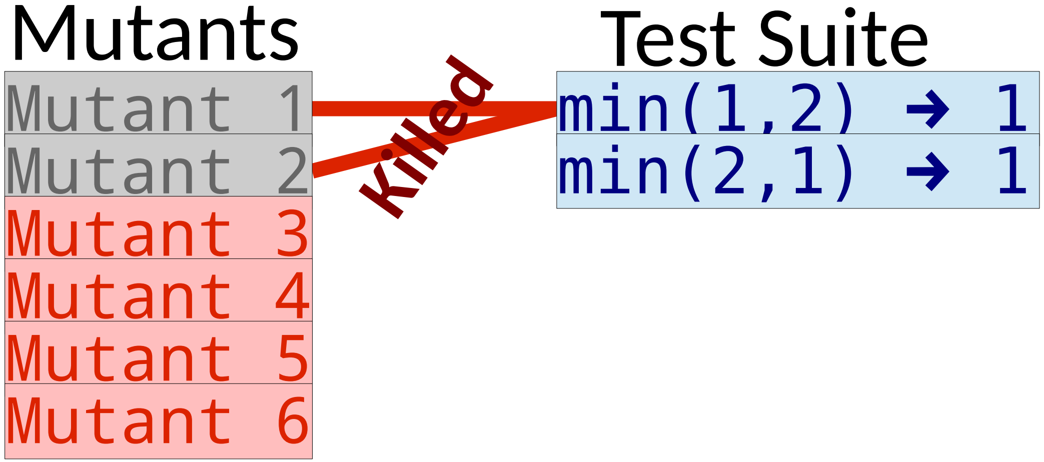
```
Mutant 1
Mutant 2
Mutant 3
Mutant 4
Mutant 5
Mutant 6
```

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

Try every mutant on test 1.

Mutation Analysis



Mutants

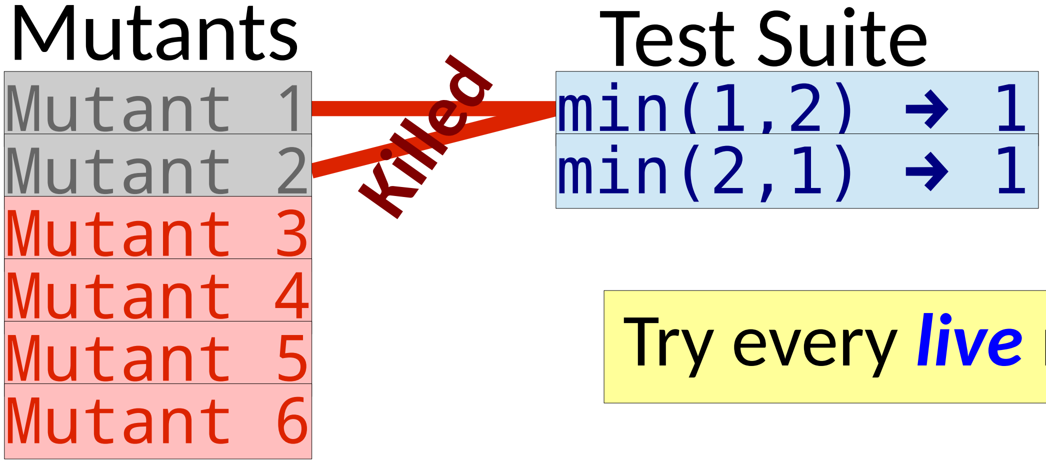
Mutant 1
Mutant 2
Mutant 3
Mutant 4
Mutant 5
Mutant 6

Killed

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

Mutation Analysis



Mutants

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

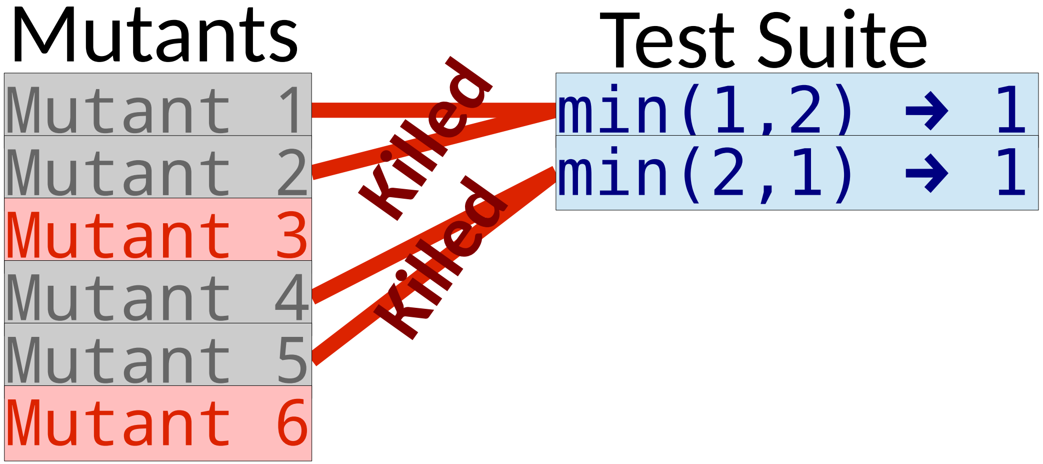
Killed

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

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

Mutation Analysis



Mutants

Mutant 1

Mutant 2

Mutant 3

Mutant 4

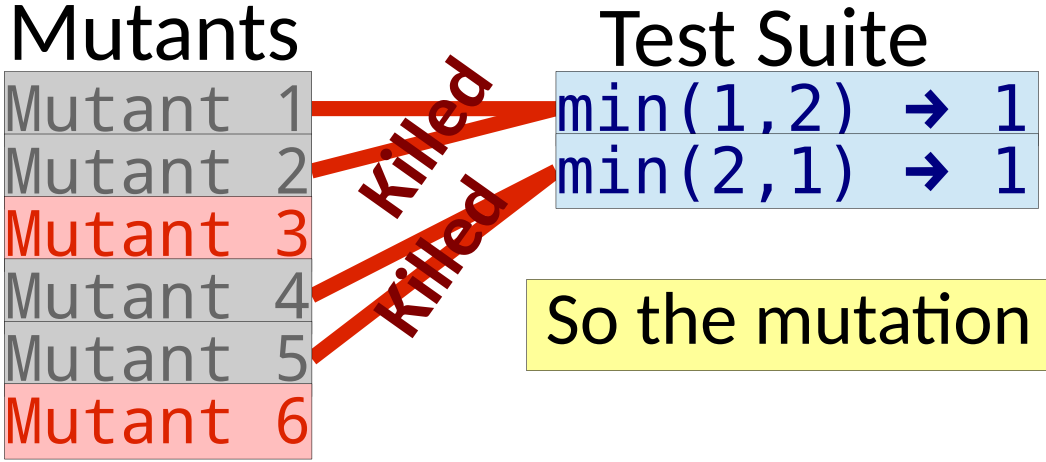
Mutant 5

Mutant 6

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

Mutation Analysis



Mutants

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

Killed

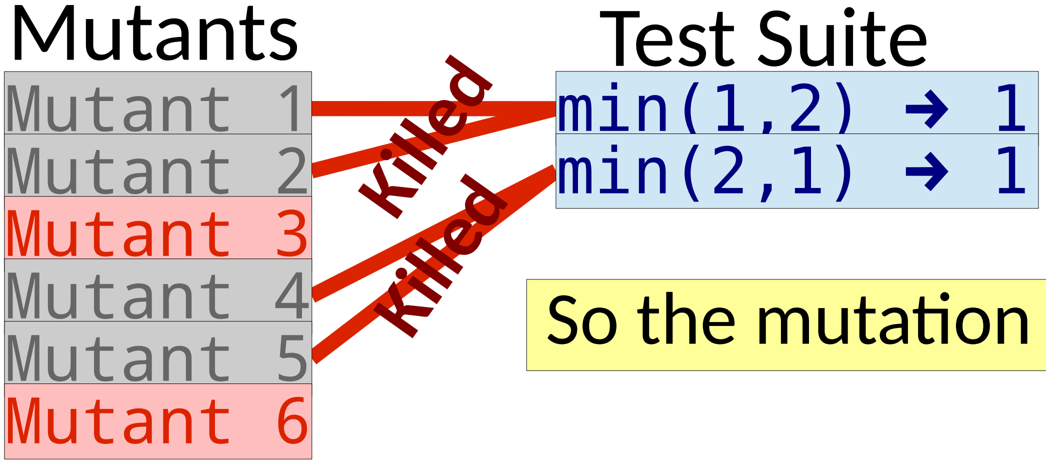
Killed

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

So the mutation score is…

Mutation Analysis



Mutants

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

Killed

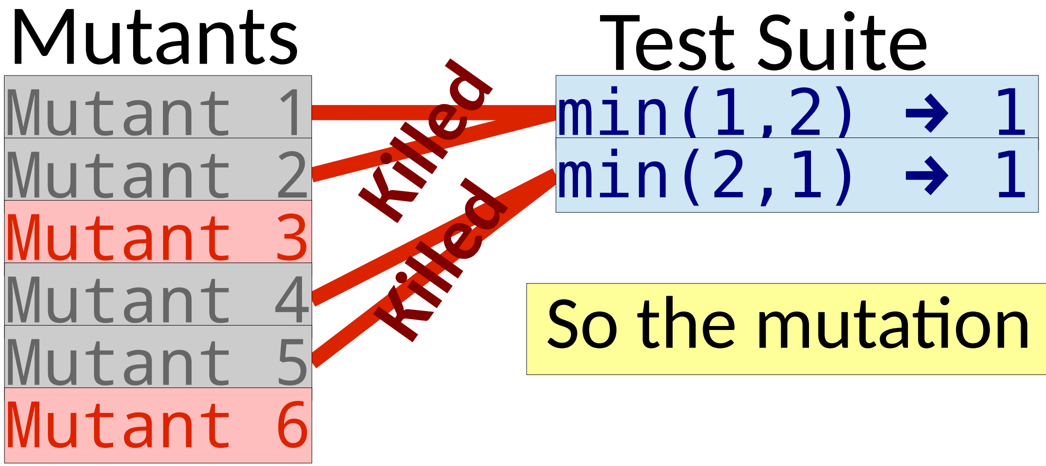
Killed

Test Suite

```
min(1,2) → 1
min(2,1) → 1
```

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

Mutation Analysis



Mutants

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

Killed

Killed

Test Suite

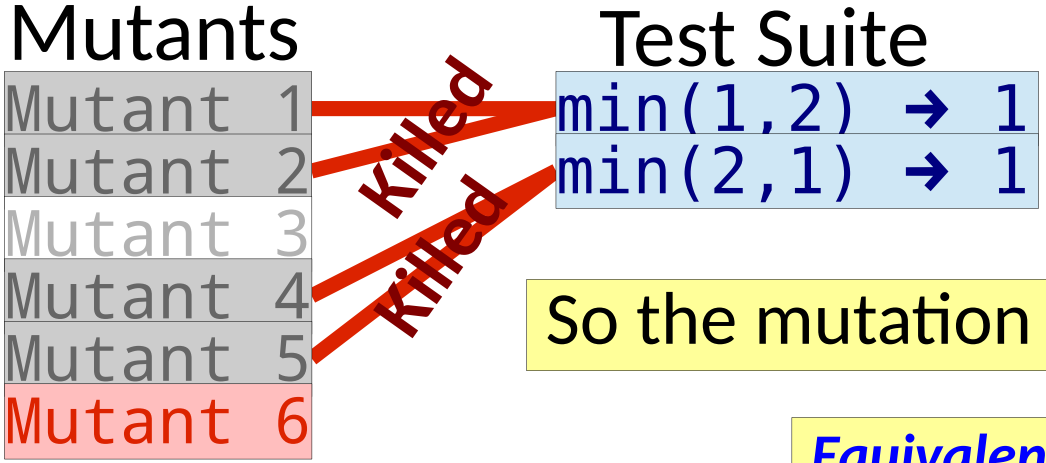
min(1,2) → 1
min(2,1) → 1

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

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

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

Mutation Analysis



Mutants

Mutant 1

Mutant 2

Mutant 3

Mutant 4

Mutant 5

Mutant 6

Test Suite

min(1,2) → 1

min(2,1) → 1

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

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

Equivalent to the original!
There is no injected bug.

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

Equivalent Mutants

- Equivalent mutants are not bugs and should not be counted

Equivalent Mutants

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

Equivalent Mutants

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

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

Start with the 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 handling of duplicate & equivalent mutants

Equivalent Mutants

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

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

- Detecting equivalent mutants is *undecidable* in general

Equivalent Mutants

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

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

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

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

Equivalent Mutants

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

Equivalent Mutants

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

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

Requires reasoning about why the result was the same.

Mutation 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.
  - Covstmt(T1) > Covstmt(T2) → ?

Traditional Coverage vs Mutation

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

Is T1 *more likely* to find more bugs?

Traditional Coverage vs Mutation

- Statement & branch based coverage are the most popular adequacy measures in practice.
  - Covstmt(T1) > Covstmt(T2) → ?

Is T1 *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.
  - Covstmt(T1) > Covstmt(T2) →

- 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.
  - Covstmt(T1) > Covstmt(T2) →

- 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.
  - Covstmt(T1) > Covstmt(T2) →

- 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.
  - Covstmt(T1) > Covstmt(T2) →

- 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.

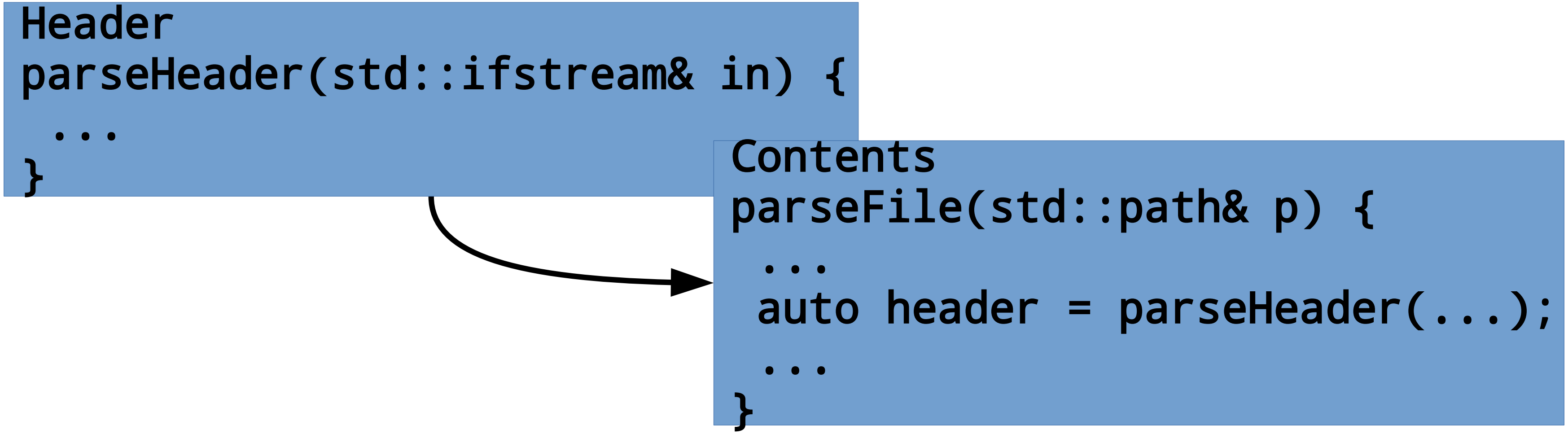
Why Use Regression Testing

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

```
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.

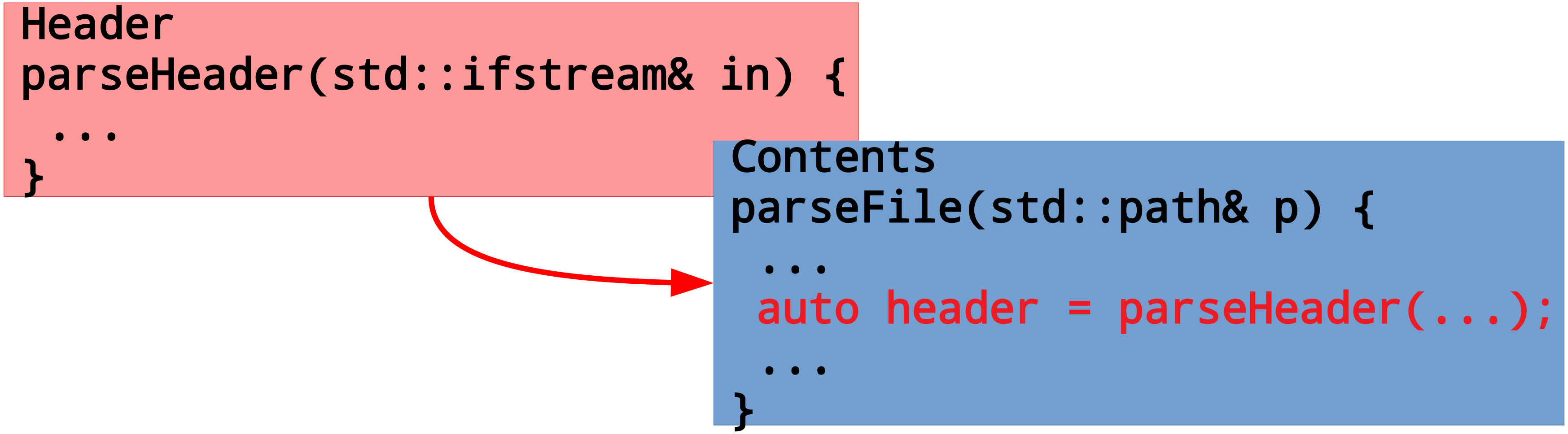


```
Header
parseHeader(std::ifstream& in) {
  ...
}
```

```
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.



```
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** (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 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 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

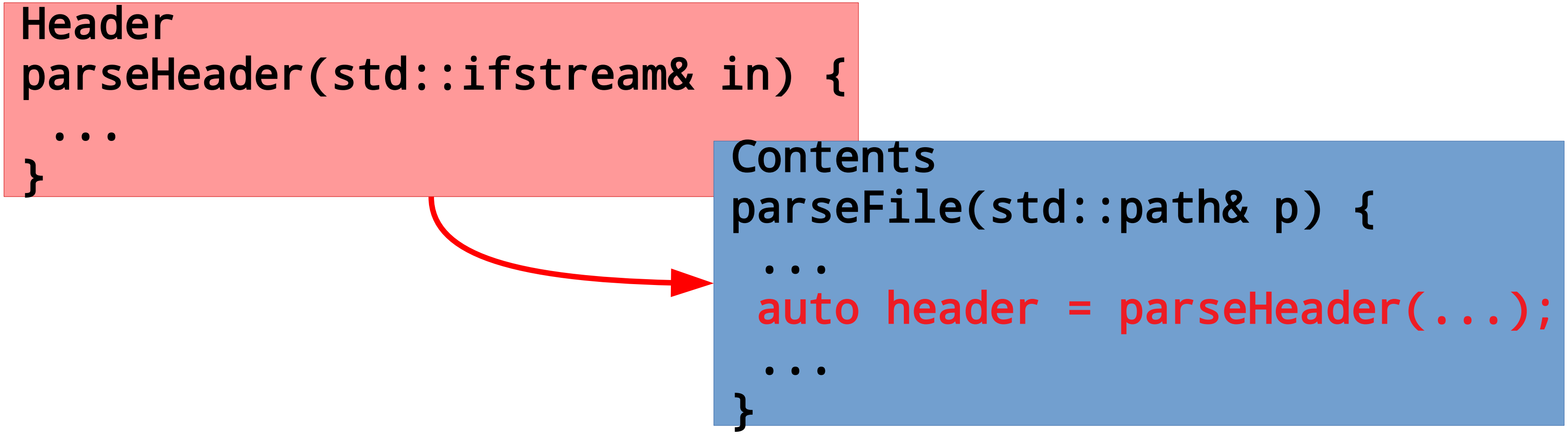
In practice, tools can assist in finding out which tests need to be run

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*

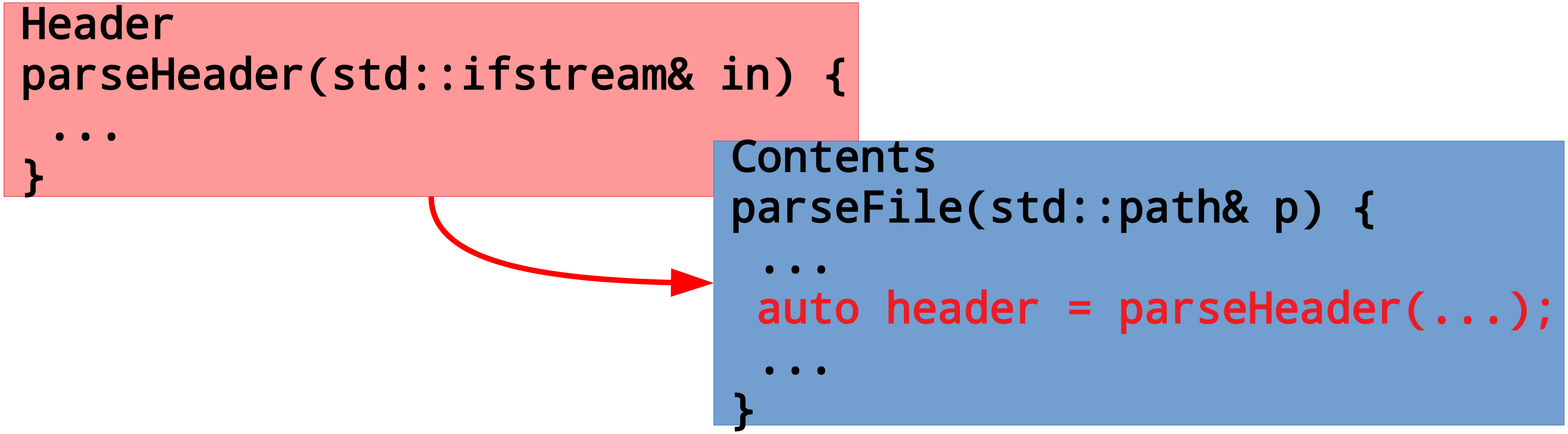


```
Header
parseHeader(std::ifstream& in) {
 ...
}
```

```
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



```
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*

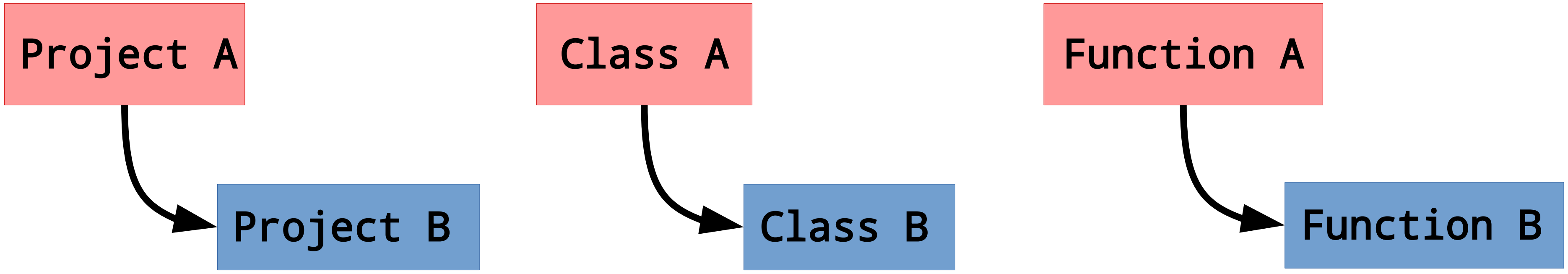
```
void
foo() {
  ...
  out = fopen(“channel.txt”,”w”);
  fwrite(out, ...);
}
```

```
void
bar() {
 ...
 in = fopen(“channel.txt”, “r”);
 fread(in, ...);
}
```

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*

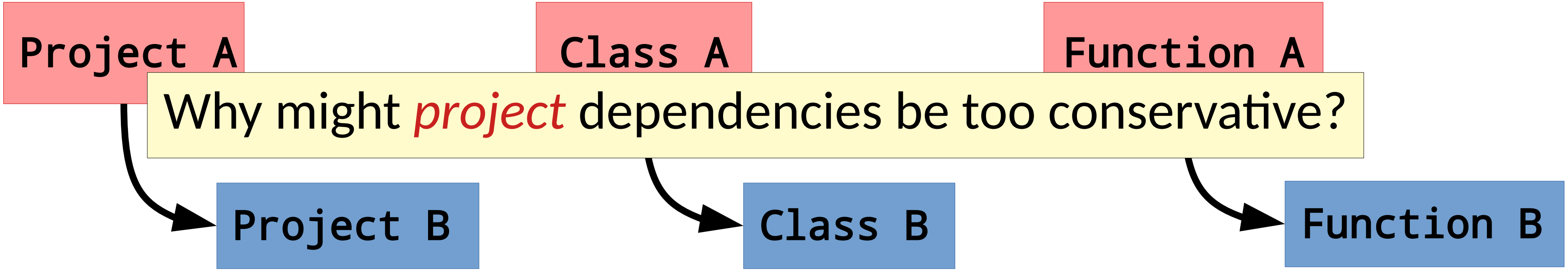
- The granularity of the analysis also affects all aspects of performance



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*

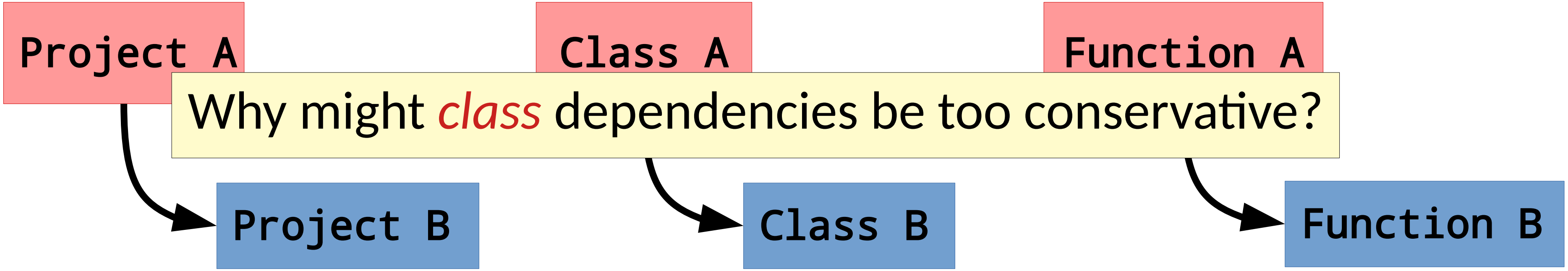
- The granularity of the analysis also affects all aspects of performance



Why might *project* dependencies be too conservative?

Change Impact Analysis & Regression Test Selection

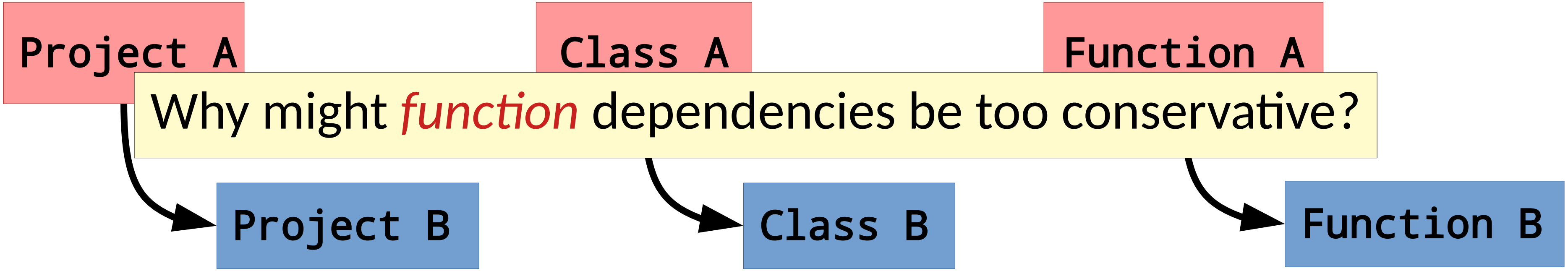
- Given a set of changes,
  regression test selection determines which tests to execute
  - The analysis detects dependencies between components
  - Only tests for components (transitively) dependent on a change need to run
  - Different forms of dependence impact the *efficiency*, *safety*, & *reduction*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



Why might *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

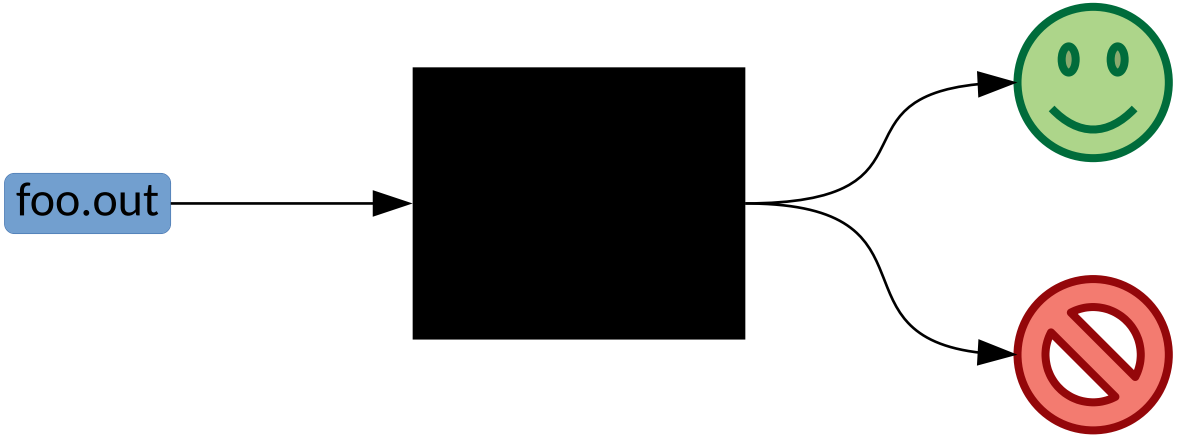
- We have considered how to
  - write tests well.
  - measure & assess a test suite.
  - efficiently & effectively add testing into a workflow.
- All of these aid using tests to know *when bugs occur*
- But we often care about other tasks:
  - *Investigating* why a bug exists
  - *Repairing* a bug
  - *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

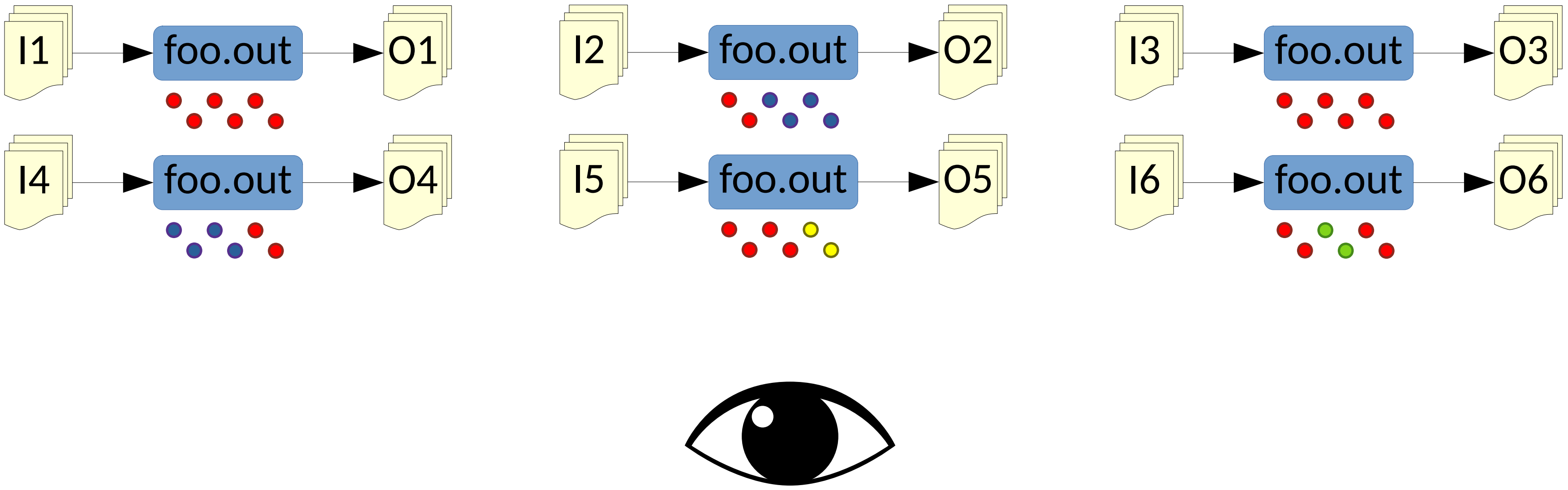
- What information does a test suite give us?
  - A weak *black box oracle* for program correctness



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 = <{ti}, o>
  - *Passing* tests p ⊂{ti}
  - *Failing* tests f ⊂{ti}
  - Observable criteria c(ti) = ci

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 = <{ti}, o>
  - Passing tests p ⊂{ti}
  - Failing tests f ⊂{ti}
  - Observable criteria c(ti) = ci
- Produce
  - A ranked list of *locations* [li] for a developer to consider

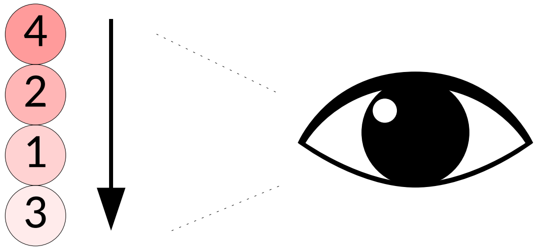
Fault Localization

```
1 if condition:
2   x = a + b
  else:
3   y = c * d
4 return x + y
```

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

Fault Localization

```
1 if condition:
2   x = a + b
  else:
3   y = c * d
4 return x + y
```



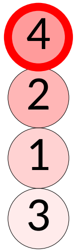
- Produce
  - A ranked list of *locations* [li] for a developer to consider

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
- 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 = <{ti}, o>
  - Passing tests p ⊂{ti}
  - Failing tests f ⊂{ti}
  - Observable criteria c(ti) = ci



- Produce
  - A ranked list of locations [li] 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

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

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

Fault Localization

T1 T2 T3 T4

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

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

Fault Localization

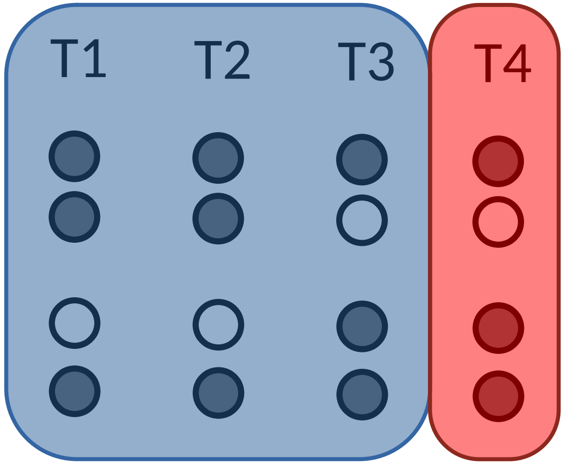
```
if condition:
  x = a + b
else:
  y = c * d
return x + y
```

T1	T2	T3	T4
●	●	●	●
●	●	○	○
○	○	●	●
●	●	●	●

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

Fault Localization

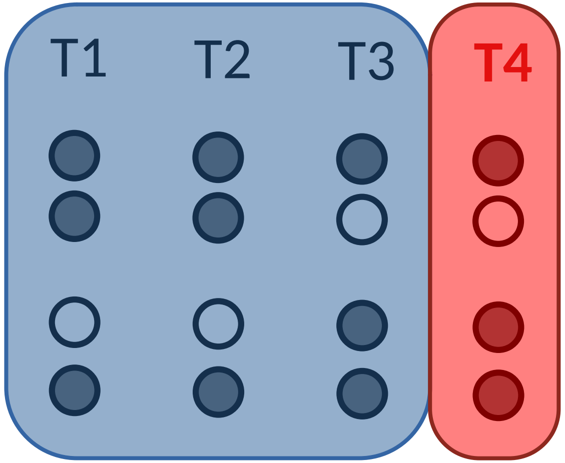
```
if condition:
  x = a + b
else:
  y = c * d
return x + y
```



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

Fault Localization

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

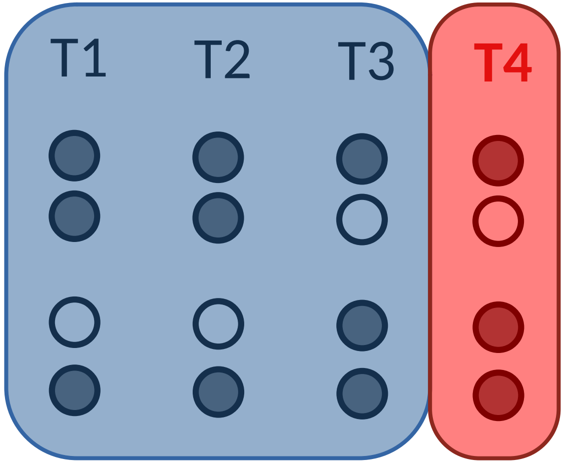


- A lot of classic techniques focus on, e.g., statement coverage

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

Fault Localization

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



- 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

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

T1	T2	T3	T4
●	●	●	●
●	●	○	○
○	○	●	●
●	●	●	●

- A lot of classic techniques focus on, e.g., statement coverage
- How should we prioritize?

Fault Localization

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

T1	T2	T3	T4
●	●	●	●
●	●	○	○
○	○	●	●
●	●	●	●

- A lot of classic techniques focus on, e.g., statement coverage
- How should we prioritize?
  - Heuristic [Jones 2005, Jiang 2019]

Fault Localization

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

T1	T2	T3	T4
●	●	●	●
●	●	○	○
○	○	●	●
●	●	●	●

$$\frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

[Tarantula]

- A lot of classic techniques focus on, e.g., statement coverage
- How should we prioritize?
  - Heuristic [Jones 2005, Jiang 2019]

Fault Localization

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

T1	T2	T3	T4
●	●	●	●
●	●	○	○
○	○	●	●
●	●	●	●

$$\frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

[Tarantula]

$P(H_i|u_i)$ Likelihood that ui caused the failure when executed

[Doric]

- A lot of classic techniques focus on, e.g., statement coverage
- How should we prioritize?
  - Heuristic [Jones 2005, Jiang 2019]
  - Statistical [Landberg 2018]

Fault Localization

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

T1	T2	T3	T4
●	●	●	●
●	●	○	○
○	○	●	●
●	●	●	●

$$\frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

[Tarantula]

$P(H_i|u_i)$ Likelihood that ui caused the failure when executed

[Doric]

- 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]
  - …

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)

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?

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

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 [δ_i] that make T pass

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!

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: [l_i]
- Produce
  - A ranked list of patches/diffs [δ_i] 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?

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]

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?

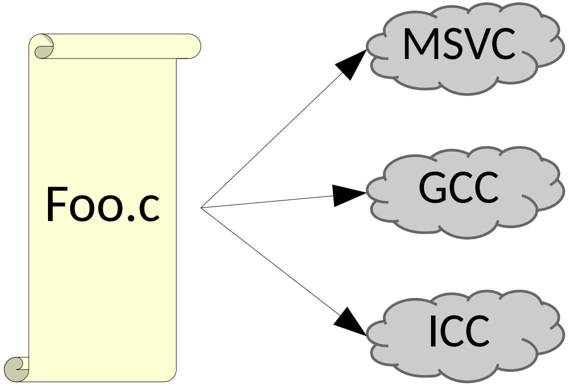
- Many compiler bugs come from "middle end" optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations

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*

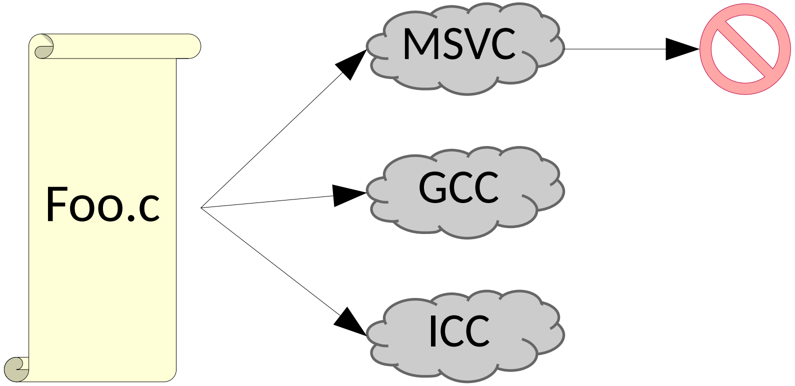
How Would You Test a Compiler?

- 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*



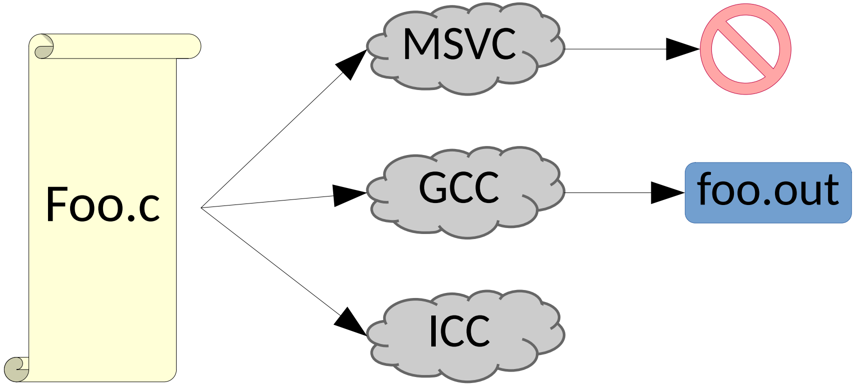
How Would You Test a Compiler?

- 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*



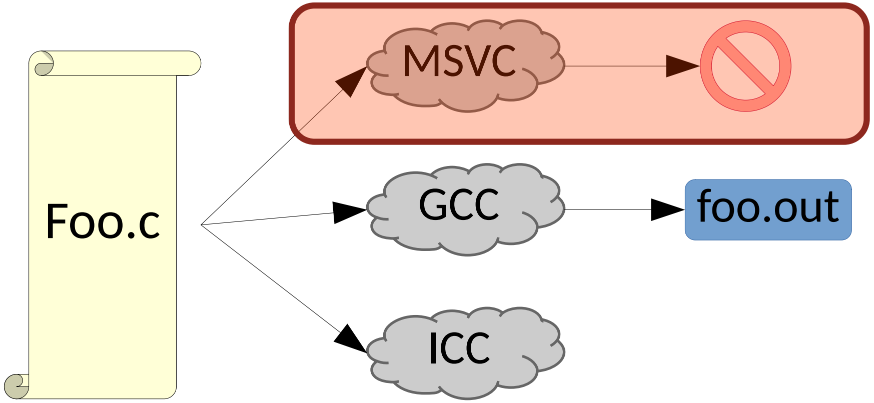
How Would You Test a Compiler?

- 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*



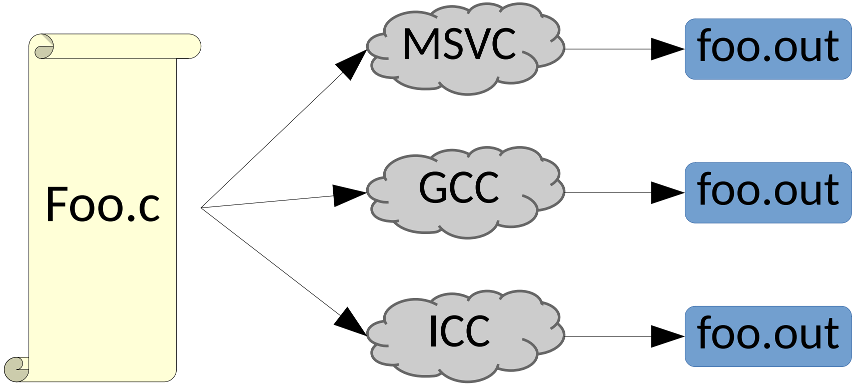
How Would You Test a Compiler?

- 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*



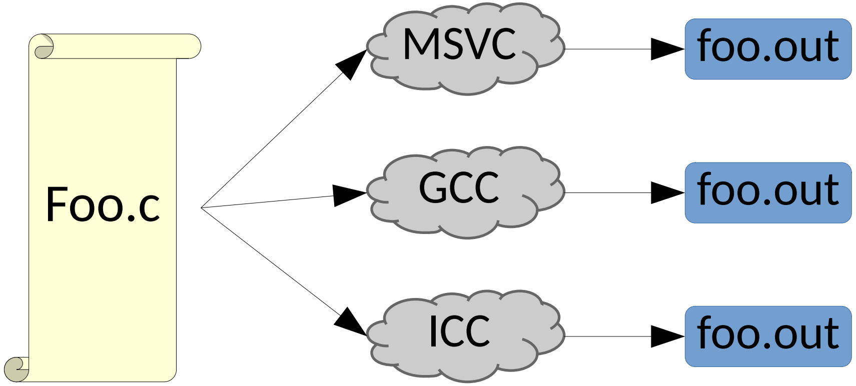
How Would You Test a Compiler?

- 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?

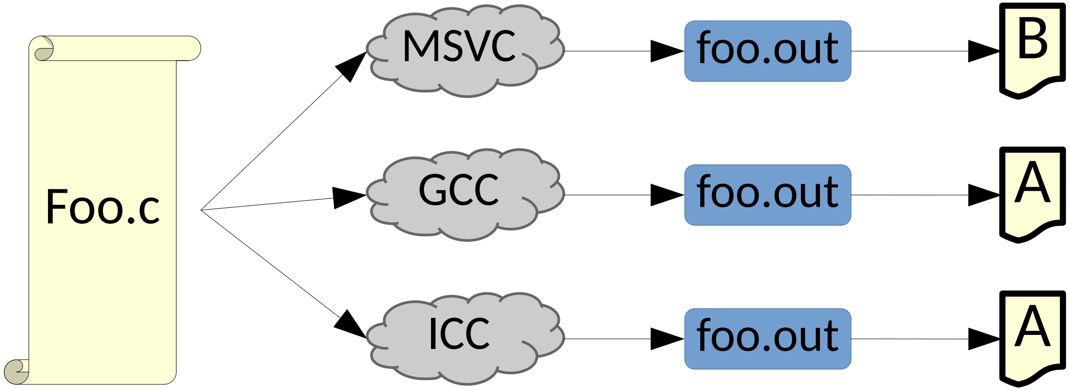
- Many compiler bugs come from “middle end” optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use *differential testing*



How might we test them here?

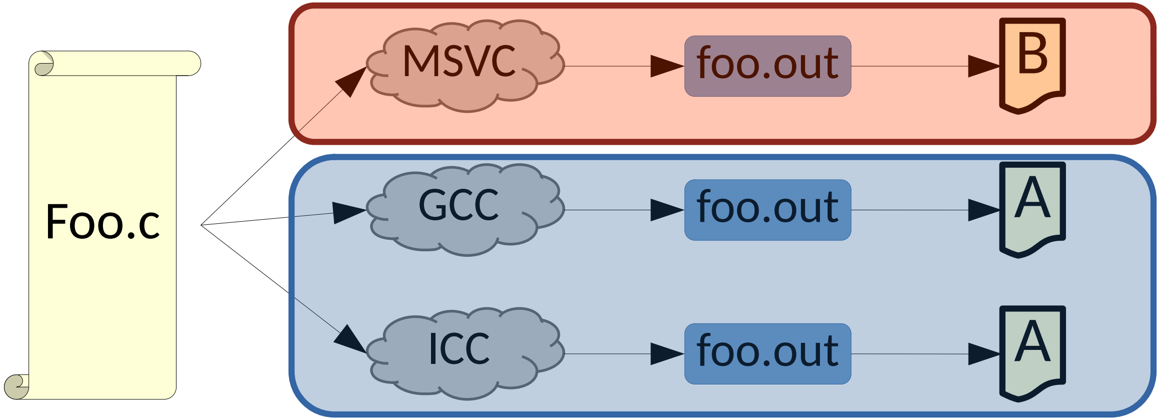
How Would You Test a Compiler?

- 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?

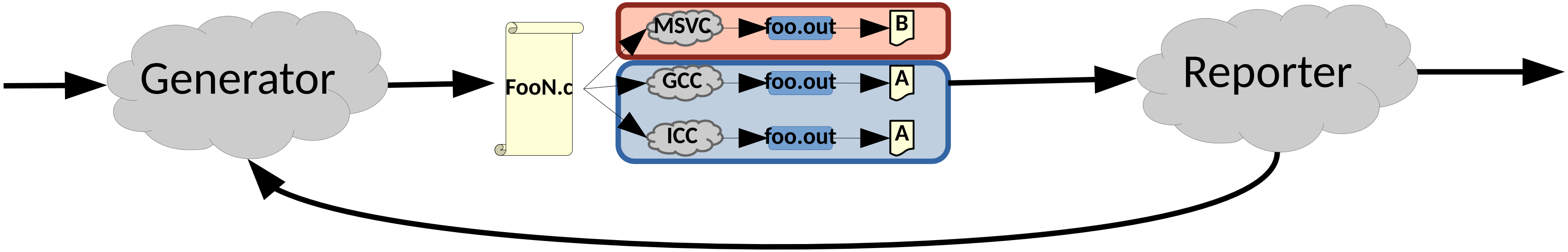
- Many compiler bugs come from "middle end" optimizations
  - Complex interactions from multiple rules make testing challenging
- But mainstream languages tend to have multiple implementations
- Big picture: use *differential testing*
  - Given compilers c1, c2, ..., ci
  - Provide an input I to each 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 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 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*

```
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

```
int x = INT_MAX;
x = x + 1;
```

- But mainstream languages tend to have multiple implementations
- Big picture: use *differential testing*

```
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

```
int x = INT_MAX;
x = x + 1;
```

- But mainstream languages tend to have multiple implementations
- Big picture: use *differential testing*

```
printf(“%p”, &someVariable);
```

  - Provide an input I to each and determine corre
  - Generate many inputs I and assess automatica
- To do that, we need to be very careful with
  - Programs should produce the same results: [Y
    deterministic
    well defined

```
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 = {(I1,O1), (I2,O2), ..., (In,On)}

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 = {(I1,O1), (I2,O2), ..., (In,On)}
- Produce
  - a test Tn+1 = F({I1,I2, ...,In}, {O1,O2, ...,On})

Pressing Further

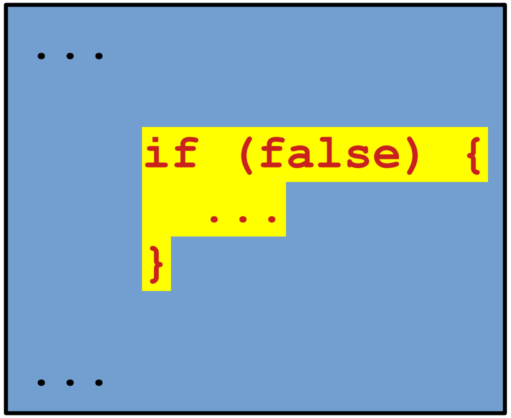
- Is there a way to get more value out of each generated test?
- Given
  - Some test T with oracle O
  - Can we produce many more tests?
- Metamorphic Testing
  - We can generate new tests using the known behavior of existing tests
- Given
  - a sequence of tests T = {(I1,O1), (I2,O2), ..., (In,On)}
- Produce
  - a test T_{n+1} = F({I1,I2, ...,In}, {O1,O2, ...,On})
- 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]

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]



```
...

    if (false) {
      ...
    }

...
```

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]

```
...

    if (false) {
      ...
    }


...
```

```
// x is profiled as < 0
...
    if (x > 0) {
      ...
    }


...
```

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]

```
...

    if (false) {
      ...
    }


...
```

```
// x is profiled as < 0
...
    if (x > 0) {
      ...
    }


...
```

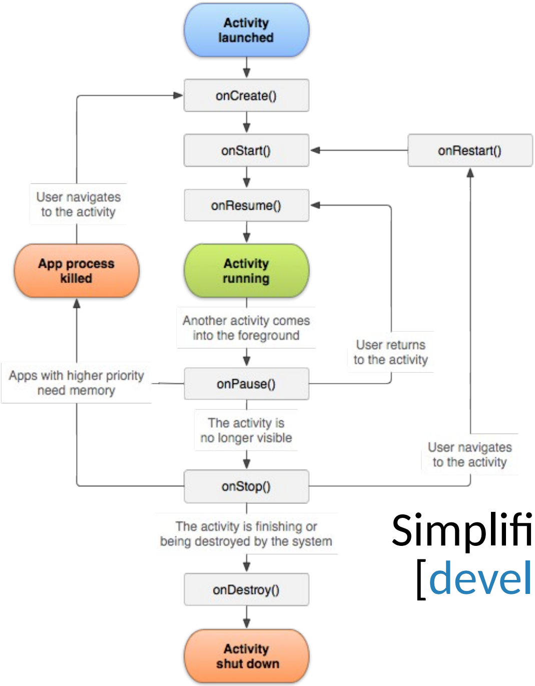
- This may seem simple, but it provides a great deal of value today
  - GCC, Clang, MSVC, ICC
  - Vulcan & OpenGL shaders
  - ...

Other Examples of Metamorphic Testing

- Android apps have complex life cycles and often experience UI glitches

Other Examples of Metamorphic Testing

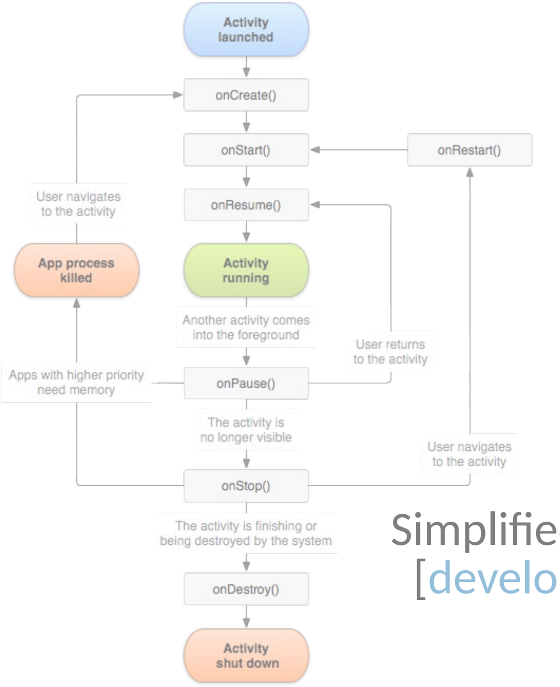
- Android apps have complex life cycles and often experience UI glitches
  - Events come in from the framework
  - Apps need to respond consistently & intuitively


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Simplified Activity Lifecycle
[developer.android.com]

Other Examples of Metamorphic Testin

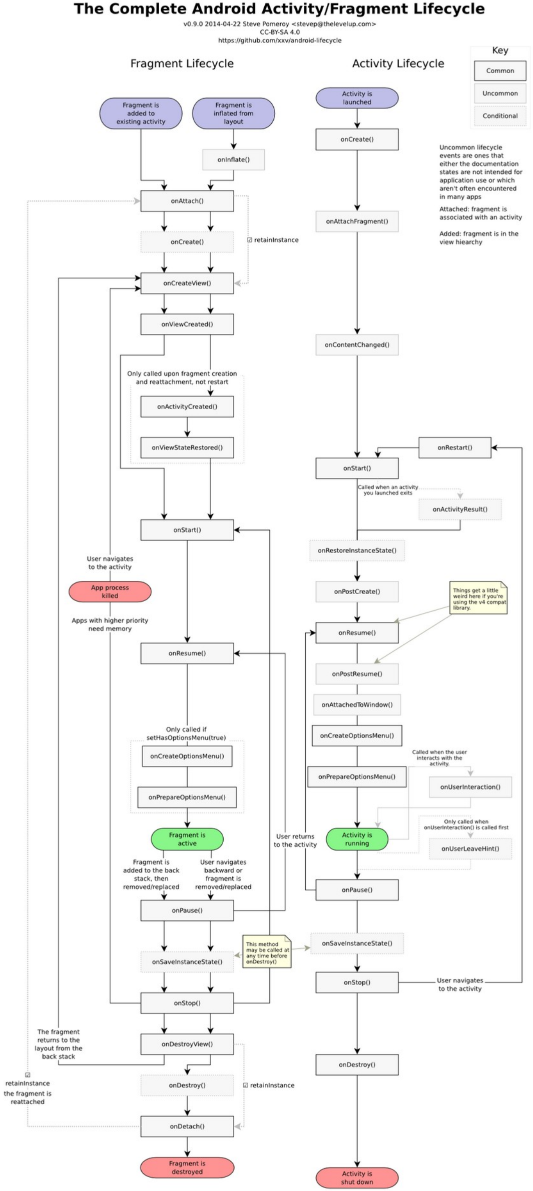
- Android apps have complex life cycles and often ex
  - Events come in from the framework
  - Apps need to respond consistently & intuitively



Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Fragment/Activity Lifecycle
[Pomeroy 2014]

Simplified Activity Lifecycle
[developer.android.com]



The Complete Android Activity/Fragment Lifecycle
v0.9.0 2014-04-22 Steve Pomeroy <stevep@thelevelup.com>
CC-BY-SA 4.0
https://github.com/xxv/android-lifecycle
Fragment Lifecycle
Activity Lifecycle
Key
Common
Uncommon
Conditional
Fragment is added to existing activity
Fragment is inflated from layout
onInflate()
onAttach()
onCreate()
☑ retainInstance
onCreateView()
onViewCreated()
Only called upon fragment creation and reattachment, not restart
onActivityCreated()
onViewStateRestored()
onStart()
User navigates to the activity
App process killed
Apps with higher priority need memory
onResume()
Only called if setHasOptionsMenu(true)
onCreateOptionsMenu()
onPrepareOptionsMenu()
Fragment is active
Fragment is added to the back stack, then removed/replaced
User navigates backward or fragment is removed/replaced
User returns to the activity
onPause()
onSaveInstanceState()
This method may be called at any time before onDestroy()
onStop()
The fragment returns to the layout from the back stack
onDestroyView()
☑ retainInstance the fragment is reattached
onDestroy()
☑ retainInstance
onDetach()
Fragment is destroyed
Activity is launched
onCreate()
Uncommon lifecycle events are ones that either the documentation states are not intended for application use or which aren't often encountered in many apps
Attached: fragment is associated with an activity
Added: fragment is in the view hiearchy
onAttachFragment()
onContentChanged()
onRestart()
onStart()
Called when an activity you launched exits
onActivityResult()
onRestoreInstanceState()
onPostCreate()
Things get a little weird here if you're using the v4 compat library.
onResume()
onPostResume()
onAttachedToWindow()
onCreateOptionsMenu()
Called when the user interacts with the activity.
onPrepareOptionsMenu()
onUserInteraction()
Activity is running
Only called when onUserInteraction() is called first
onUserLeaveHint()
onPause()
onSaveInstanceState()
onStop()
User navigates to the activity
onDestroy()
Activity is shut down

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

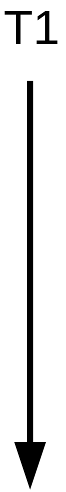
- Android apps have complex life cycles and often experience UI glitches
  - Events come in from the framework
  - Apps need to respond consistently & intuitively
  - Handling events poorly leads to:
    - Crashes
    - Non-responsiveness
    - Unexpected UI changes
    - …

Again, metamorphic testing makes this simpler.
Ideas?

Other Examples of Metamorphic Testing

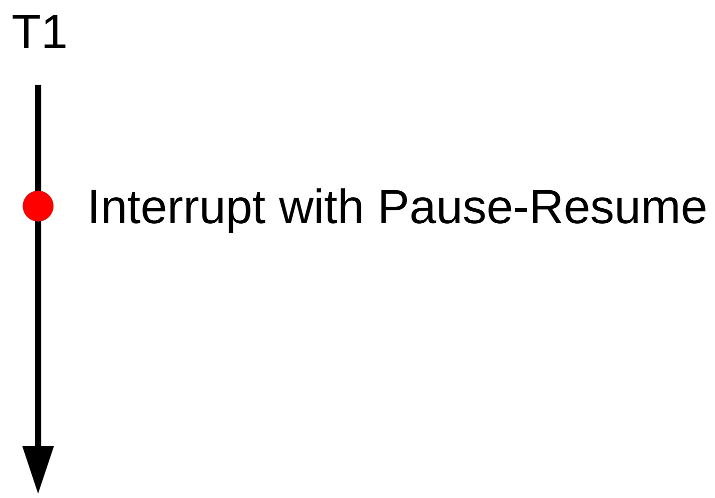
- Android apps have complex life cycles and often experience UI glitches
  - Events come in from the framework
  - Apps need to respond consistently & intuitively
  - Handling events poorly leads to:
    - Crashes
    - Non-responsiveness
    - Unexpected UI changes
    - …
- Adversarial event sequences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



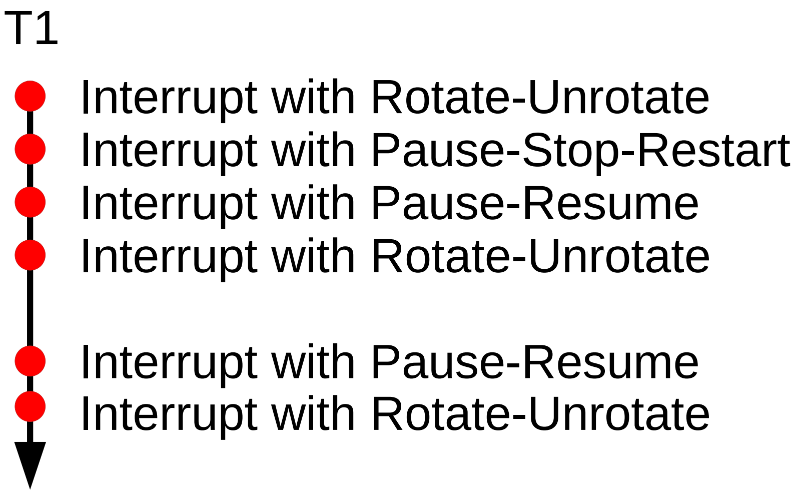
- Adversarial event sequences can just be injected into existing tests.
  - e.g. Pause-Resume, Pause-Stop-Restart, Rotate-Unrotate, …
  - Robust behavior is the same with or without these additions [Quist 2015]

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]

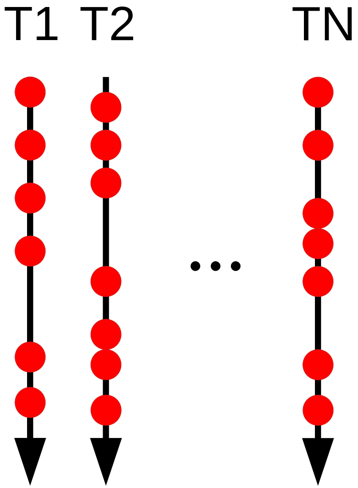
Other Examples of Metamorphic Testing



T1
Interrupt with Rotate-Unrotate
Interrupt with Pause-Stop-Restart
Interrupt with Pause-Resume
Interrupt with Rotate-Unrotate
Interrupt with Pause-Resume
Interrupt with Rotate-Unrotate

- 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

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

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

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!