Static and Dynamic Analysis of Test Suites

Patrick Lam
University of Waterloo

October 3, 2015

Goal

Convince you to
analyze *tests* (with their programs).

Why do we analyze programs?

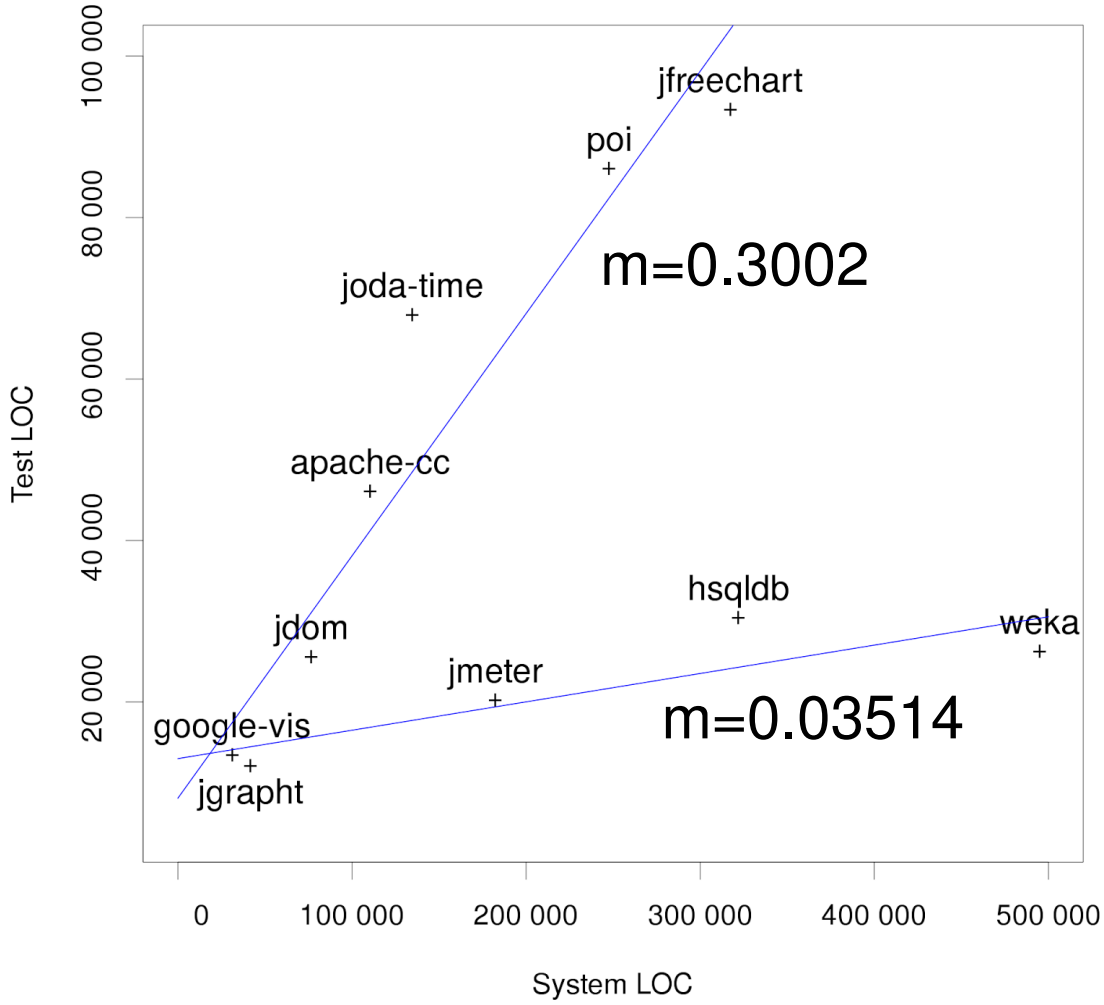
Motivations for Program Analysis

- enable program understanding & transformation;
- eliminate bugs;
- ensure software quality.

Observation

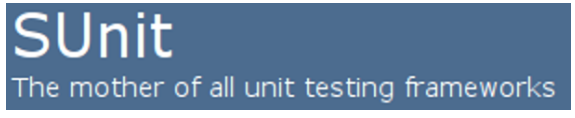
Programs come with gobs of tests.

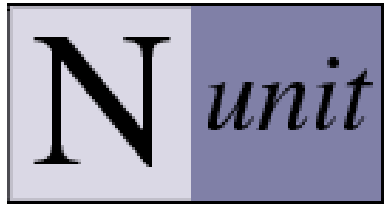
Pervasiveness of Test Suites

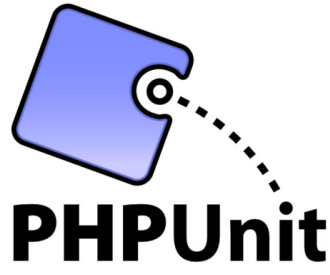


Tests and Modern Software

Extensive library support:







What Tests Do

Tests encode:

- how to invoke the system under test; and,
- what it should do.

Opportunity

Leverage test suites in program analysis.

Related Work

Static Analysis Limitations

Dynamic Analysis Limitations

Concolic analysis

Run the program with arbitrary inputs,
(some symbolic);
use solver to find new paths/inputs.

PHP Analysis (Kneuss, Suter, Kuncak)

- Choose program inputs and run the program.
- Observe the configuration loading phase.
- Use configuration information to do type analysis on the remainder of the program.

TamiFlex (Bodden et al)

- Choose program inputs and run the program.
- Observe classes loaded through reflection and custom classloaders.

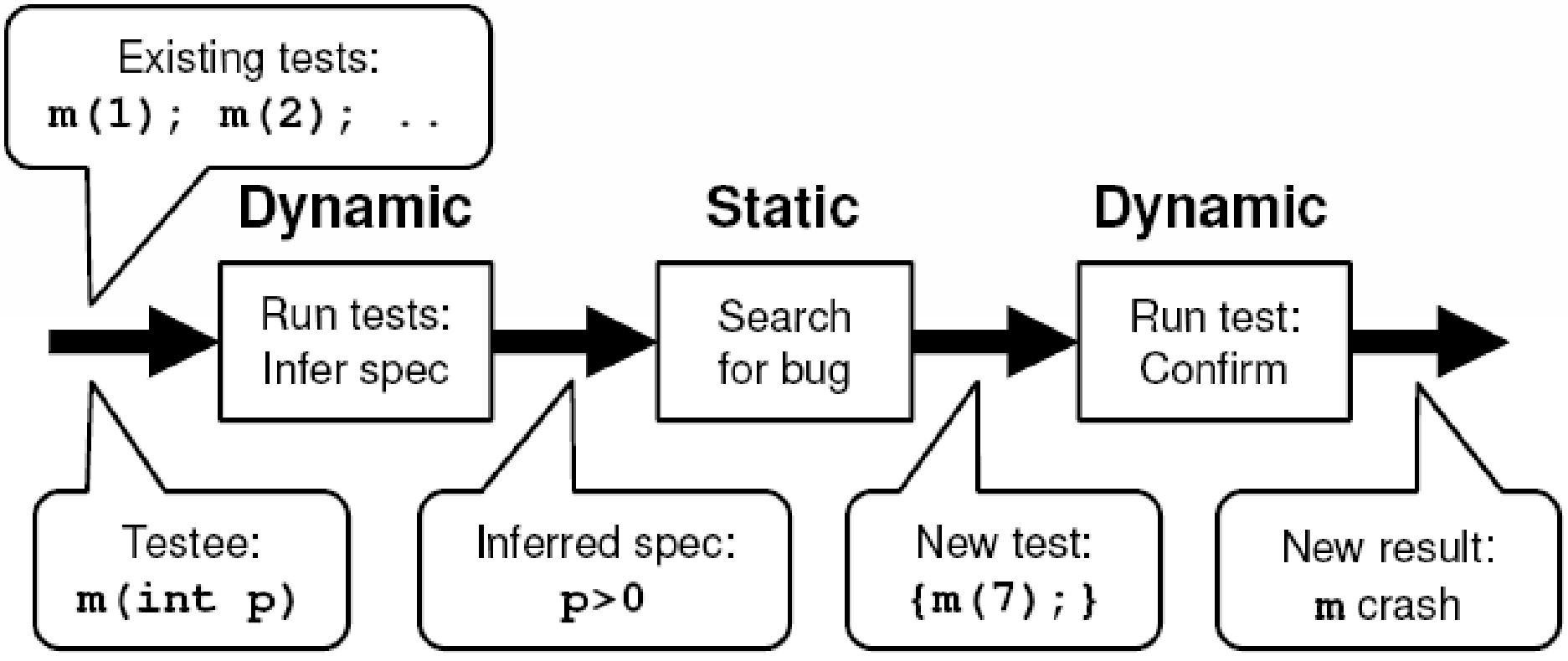
Dependent Array Type Inference from Tests (Zhu, Nori & Jagannathan)

Goal: learn quantified array invariants.

Approach: observe from test runs; namely:

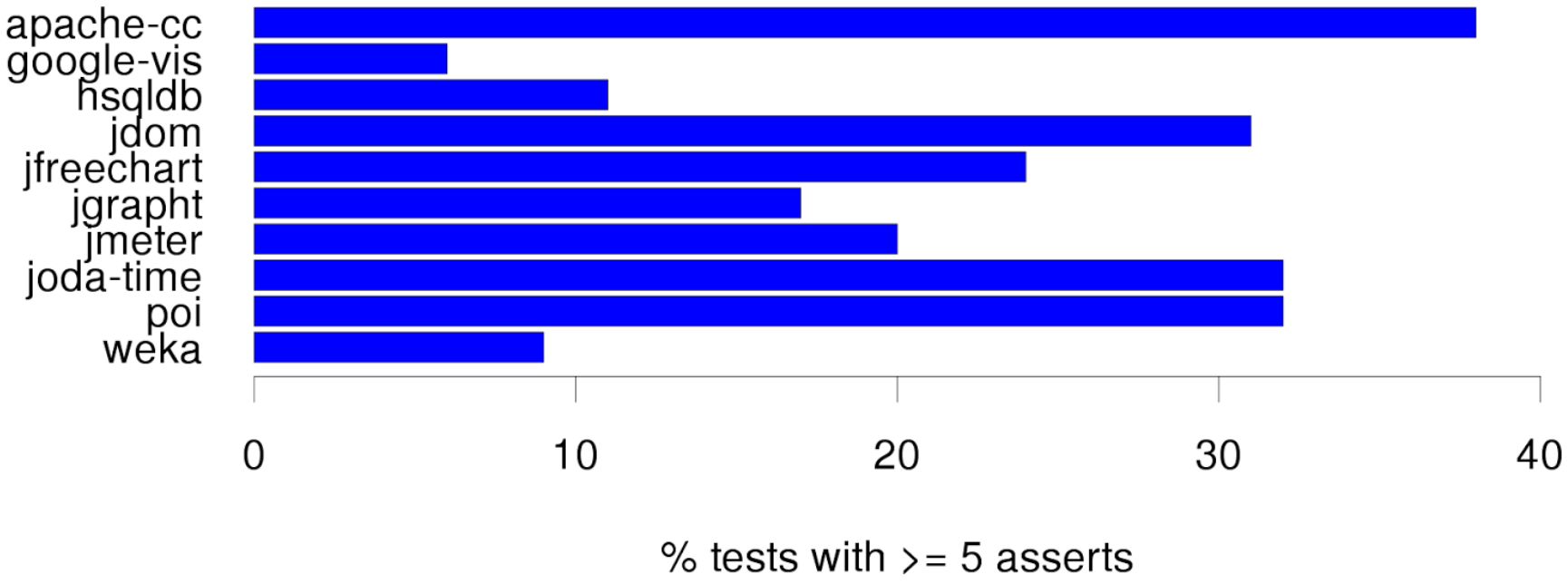
- guess coarse templates;
- run with simple random tests;
- generate constraints;
- validate types.

DSD-Crasher (Csallner and Smaragdakis)

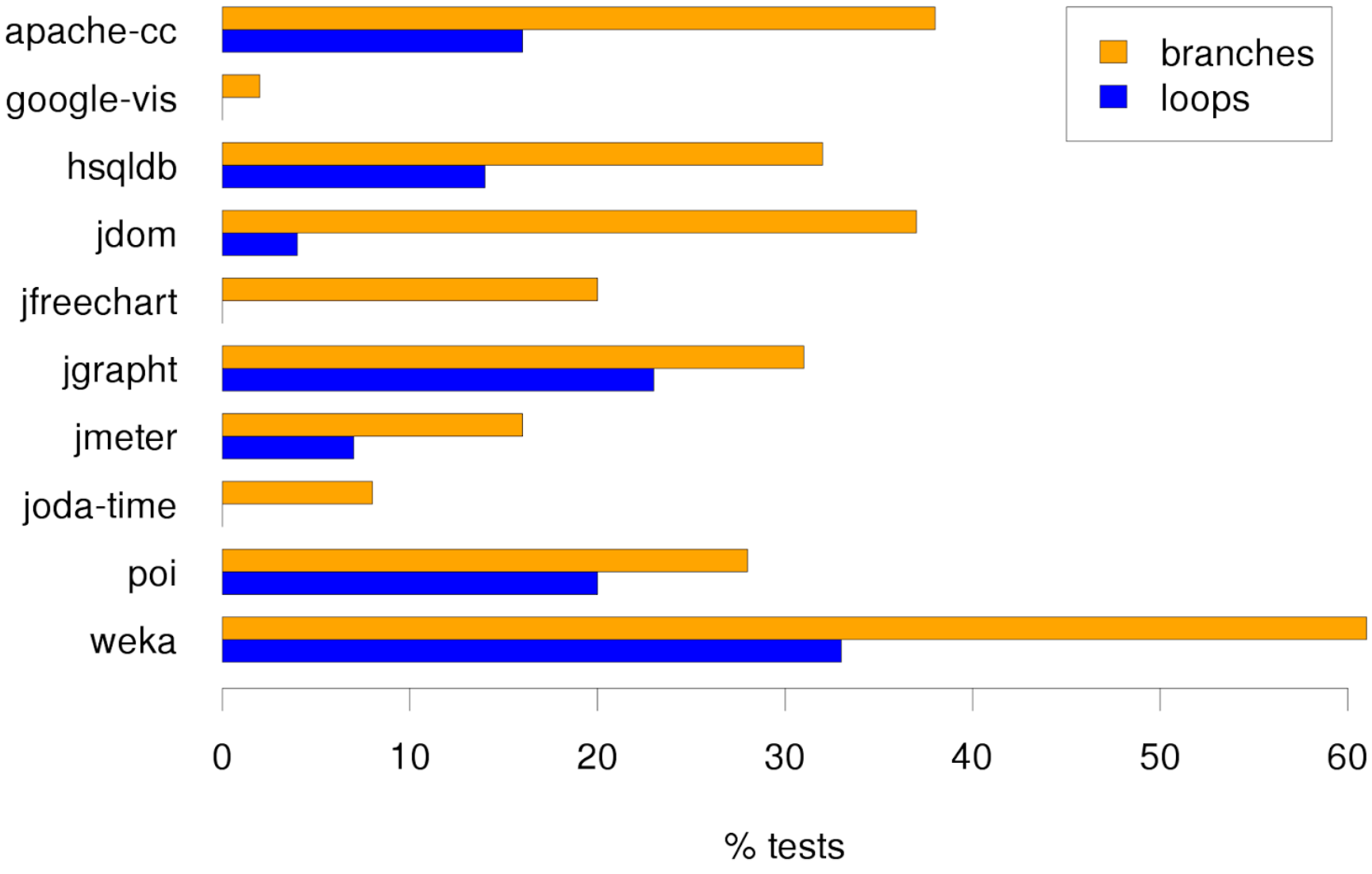


Empirical Studies

Complexity of Test Suites



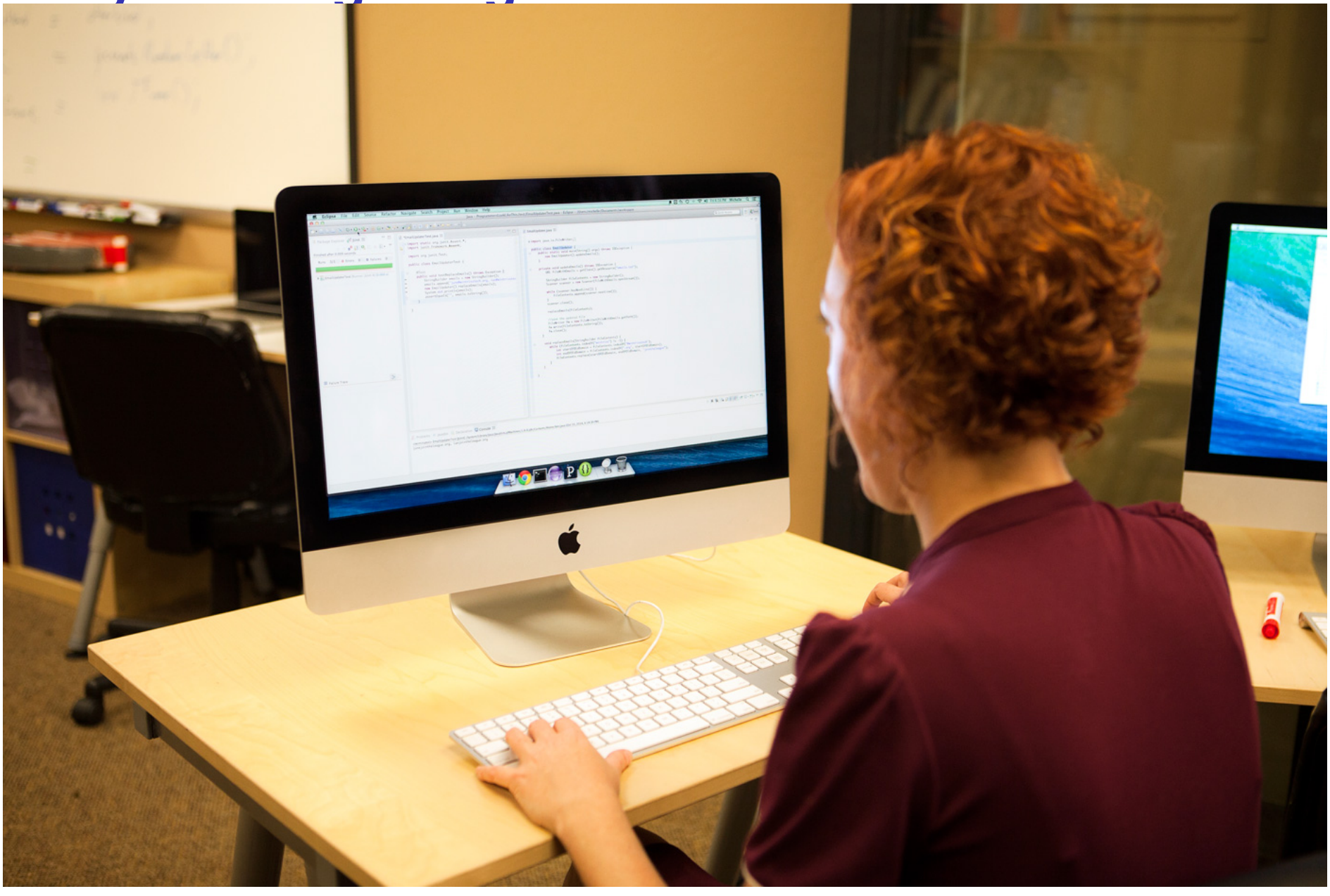
% Tests with control-flow



Similar Test Methods
(a case study in static analysis of tests)

(with Felix Fang)

Story: Writing Widget class



By Joonspoon (Own work) [CC BY-SA 4.0 (`http://creativecommons.org/licenses/by-sa/4.0`)], via Wikimedia Commons

Story: Writing Widget class

```
1  class FooWidget extends
       Widget {
2      /*...*/
3  }
4
5  class BarWidget extends
       Widget {
6      /*...*/
7  }
```

```
1  @Test
2  class FooWidgetTest {
3      /*...*/
4  }
5
6  @Test
7  class BarWidgetTest {
8      /*...*/
9  }
```

Story: Writing New Code

```
1  class BazWidget extends
       Widget {
2      /* New Code */
3  }
```

```
1  @Test
2  class BazWidgetTest {
3      /* ? */
4  }
```

Story: Writing New Code

```
1  class BazWidget extends
       Widget {
2      /* New Code */
3  }
```

```
1  @Test
2  class BazWidgetTest {
3  /* Ctrl-C, Ctrl-V */
4  }
```

Hypothesis

- Developers often copy-paste tests (and probably enjoy doing it).
- Why? JUnit test cases tend to be self-contained.
- Test clones can become difficult to comprehend or maintain later.

Benefits of Refactoring Tests

- xUnit Test Patterns: Refactoring Test Code (Meszaros)
- Reduce long term maintenance cost (Saff)
- Can detect new defects and increase branch coverage if tests are parametrized (Thummalapenta et al)
- Reduce brittleness and improve ease of understanding

Refactoring Techniques

- Language features such as inheritance or generics
- Parametrized Unit Tests and Theories

Refactoring Example

```
1  public void testNominalFiltering() {
2    m_Filter = getFilter(Attribute.NOMINAL);
3    Instances result = useFilter();
4    for (int i = 0; i < result.numAttributes(); i++)
5       assertTrue(result.attribute(i).type() != Attribute.NOMINAL);
6  }
```

```
1  public void testStringFiltering() {
2    m_Filter = getFilter(Attribute.STRING);
3    Instances result = useFilter();
4    for (int i = 0; i < result.numAttributes(); i++)
5       assertTrue(result.attribute(i).type() != Attribute.STRING);
6  }
```

Refactored Example

```
1   static final int [] filteringTypes = {
2     Attribute.NOMINAL, Attribute.STRING,
3     Attribute.NUMERIC, Attribute.DATE
4   };
5
6   public void testFiltering() {
7     for (int type : filteringTypes)
8       testFiltering(type);
9   }
10
11  public void testFiltering(final int type) {
12    m_Filter = getFilter(type);
13    Instances result = useFilter();
14    for (int i = 0; i < result.numAttributes(); i++)
15      assertTrue(result.attribute(i).type() != type);
16  }
```

Our Contributions

- Test refactoring candidate detection technique using *Assertion Fingerprints*
- Empirical and qualitative analyses of results on 10 Java benchmarks

What is a test case made of?

- Setup
- Run/Exercise
- Verify
- Teardown

What is a test case made of?

- Setup
- Run/Exercise
- Verify
- Teardown

Key Insight

Similar tests often have similar sets of asserts.

Similar Sets of Asserts: Example

```
1  public void test1() {
2     /* ... */
3     assertEqual(int, int);
4     /* ... */
5     assertEqual(float, float);
6     /* ... */
7     assertTrue(boolean);
8  }
```

```
1  public void test2() {
2     /* ... */
3     assertEqual(int, int);
4     /* ... */
5     assertEqual(float, float);
6     /* ... */
7     assertTrue(boolean);
8  }
```

Assertion Fingerprints

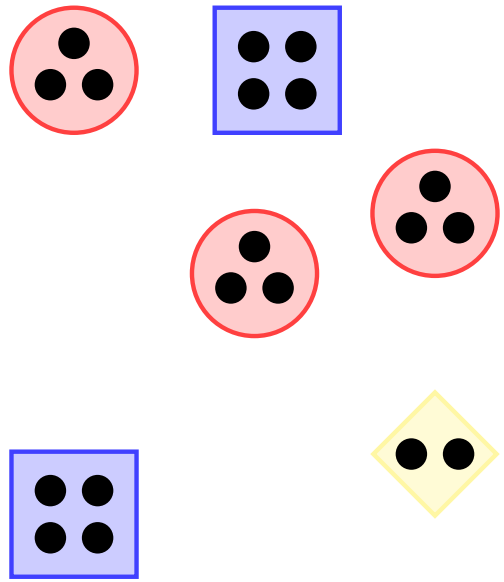
Assertion Fingerprints

Augment set of assertions.

For each assertion call, collect:

- Parameter types
- **Control flow components**

Using Assertion Fingerprints



Group tests with similar assert structures.

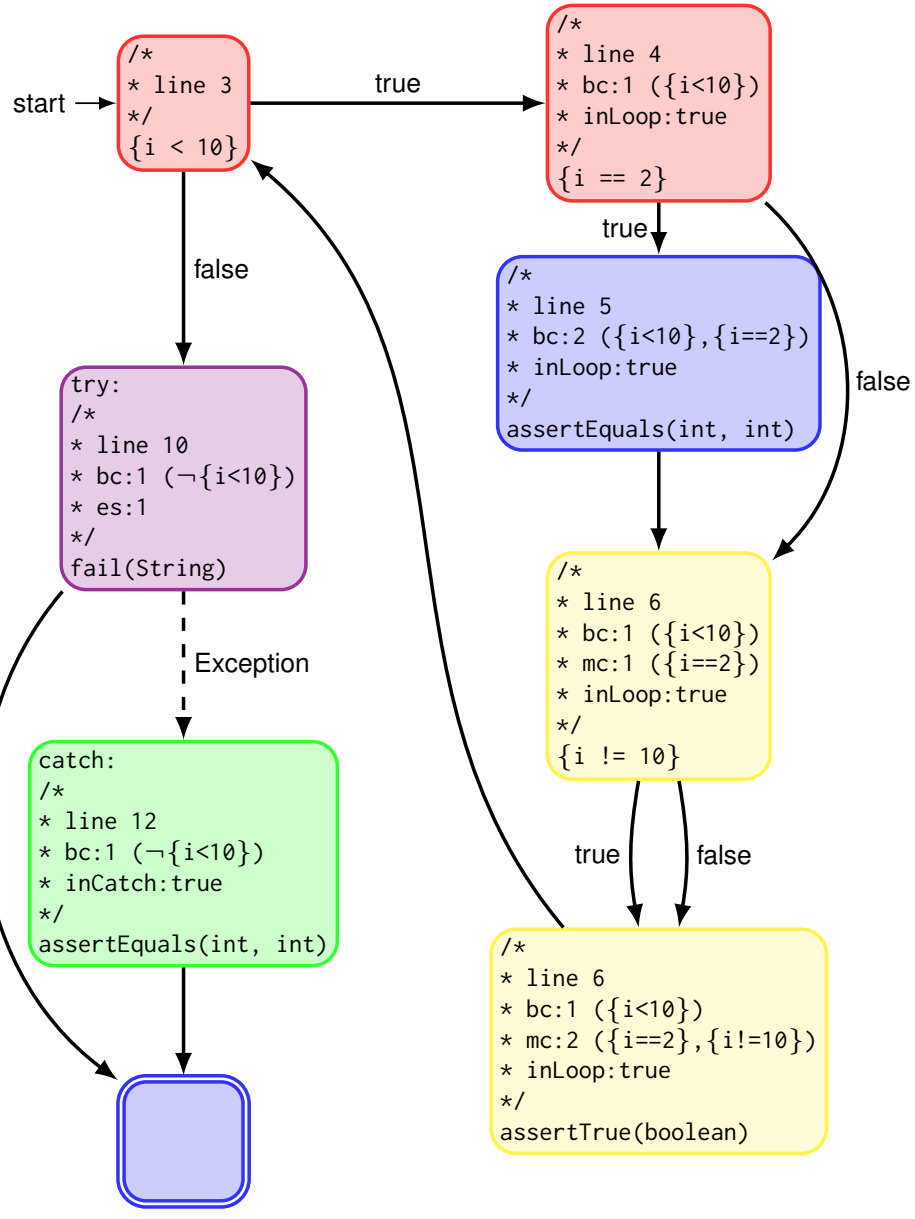
Assertion Fingerprints: Control Flow Components

- Branch Count (Branch Vertices)
- Merge Count (Merge Vertices)
- In Loop Flag (Depth First Traversal)
- Exceptional Successor Count (Exceptional CFG)
- In Catch Block Flag (Dominator Analysis)

Example: Code

```
1  public void test() {
2     int i;
3     for (i = 0; i < 10; ++i) {
4        if (i == 2)
5           assertEquals(i, 2);
6        assertTrue(i != 10);
7     }
8     try {
9        throw new Exception();
10       fail("Should have thrown exception");
11    } catch (final Exception e) {
12       assertEquals(i, 10);
13    }
14 }
```

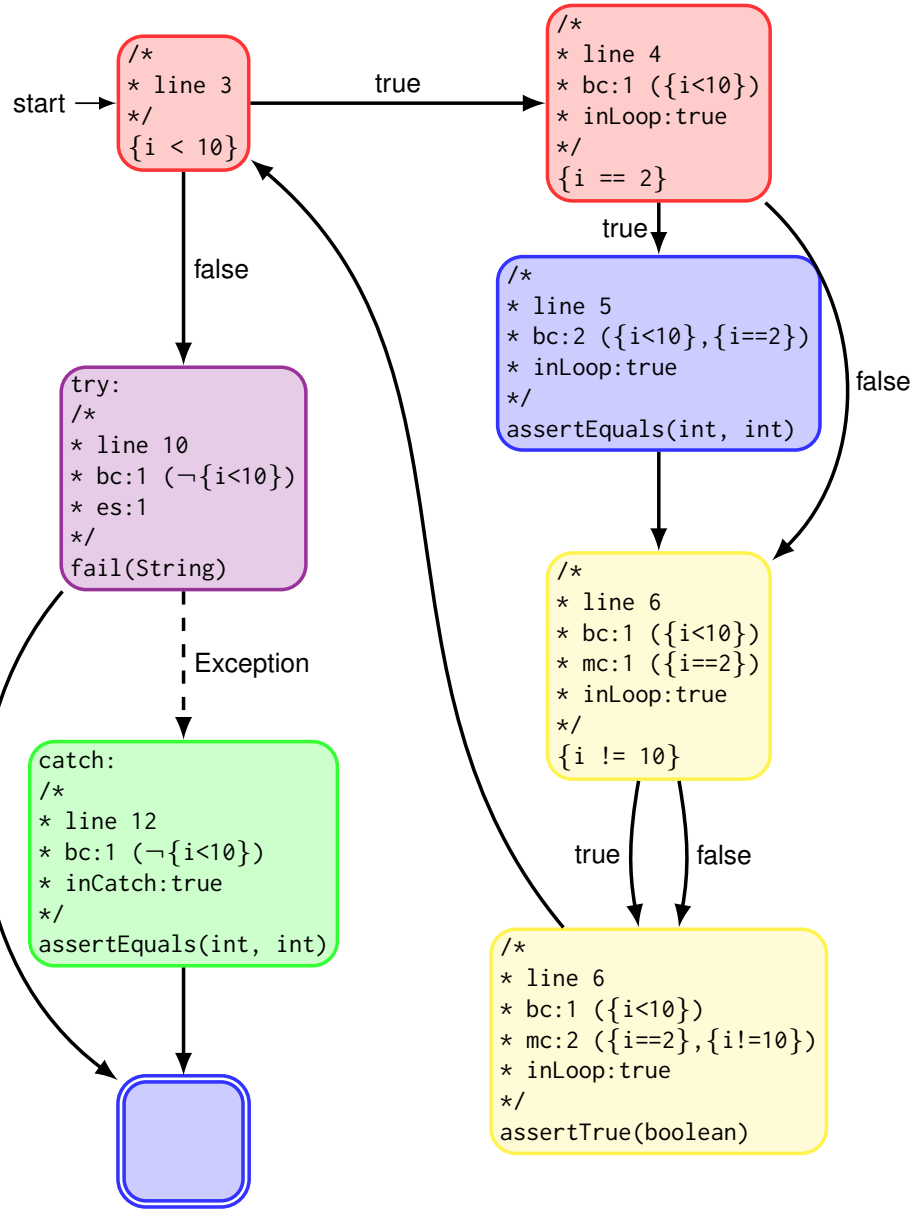
Example: CFG


start
/*
* line 3
*/
{i < 10}
true
/*
* line 4
* bc:1 ({i<10})
* inLoop:true
*/
{i == 2}
true
false
/*
* line 5
* bc:2 ({i<10},{i==2})
* inLoop:true
*/
assertEquals(int, int)
false
try:
/*
* line 10
* bc:1 (¬{i<10})
* es:1
*/
fail(String)
Exception
/*
* line 6
* bc:1 ({i<10})
* mc:1 ({i==2})
* inLoop:true
*/
{i != 10}
true
false
catch:
/*
* line 12
* bc:1 (¬{i<10})
* inCatch:true
*/
assertEquals(int, int)
/*
* line 6
* bc:1 ({i<10})
* mc:2 ({i==2},{i!=10})
* inLoop:true
*/
assertTrue(boolean)

Branch Count: Intuition

- An assertion inside an if statement is likely to be different from one that is not.

Example: CFG


start
/*
* line 3
*/
{i < 10}
true
/*
* line 4
* bc:1 ({i<10})
* inLoop:true
*/
{i == 2}
true
false
/*
* line 5
* bc:2 ({i<10},{i==2})
* inLoop:true
*/
assertEquals(int, int)
false
try:
/*
* line 10
* bc:1 (¬{i<10})
* es:1
*/
fail(String)
Exception
/*
* line 6
* bc:1 ({i<10})
* mc:1 ({i==2})
* inLoop:true
*/
{i != 10}
true
false
catch:
/*
* line 12
* bc:1 (¬{i<10})
* inCatch:true
*/
assertEquals(int, int)
/*
* line 6
* bc:1 ({i<10})
* mc:2 ({i==2},{i!=10})
* inLoop:true
*/
assertTrue(boolean)

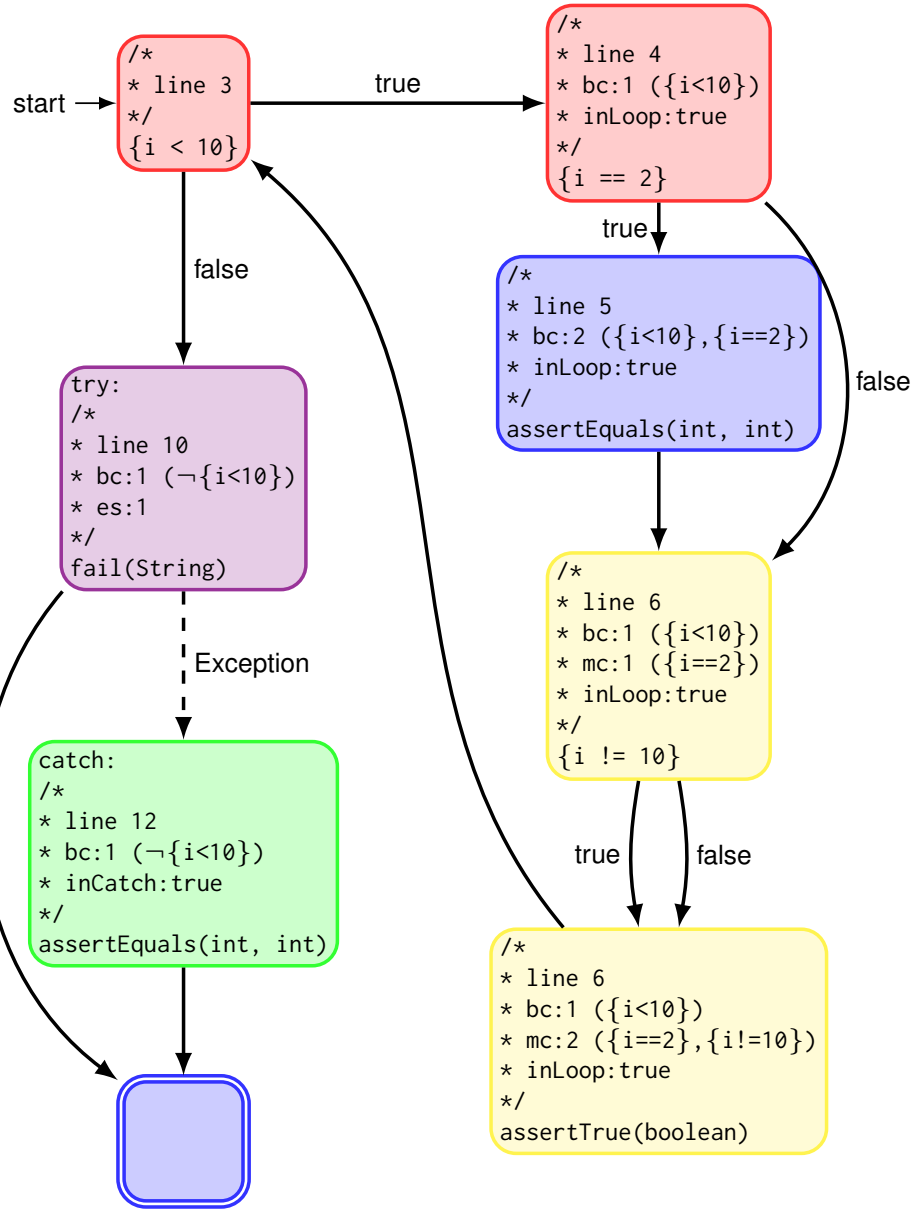
Branch Count

- Minimal number of branches needed to reach that vertex from the start of a method, excluding `n`.

Merge Count: Intuition

- An assertion with some prior operations inside an `if` statement is likely to be different from one that is not with any.

Example: CFG


start
/*
* line 3
*/
{i < 10}
true
/*
* line 4
* bc:1 ({i<10})
* inLoop:true
*/
{i == 2}
true
false
/*
* line 5
* bc:2 ({i<10},{i==2})
* inLoop:true
*/
assertEquals(int, int)
false
try:
/*
* line 10
* bc:1 (¬{i<10})
* es:1
*/
fail(String)
Exception
/*
* line 6
* bc:1 ({i<10})
* mc:1 ({i==2})
* inLoop:true
*/
{i != 10}
true
false
catch:
/*
* line 12
* bc:1 (¬{i<10})
* inCatch:true
*/
assertEquals(int, int)
/*
* line 6
* bc:1 ({i<10})
* mc:2 ({i==2},{i!=10})
* inLoop:true
*/
assertTrue(boolean)

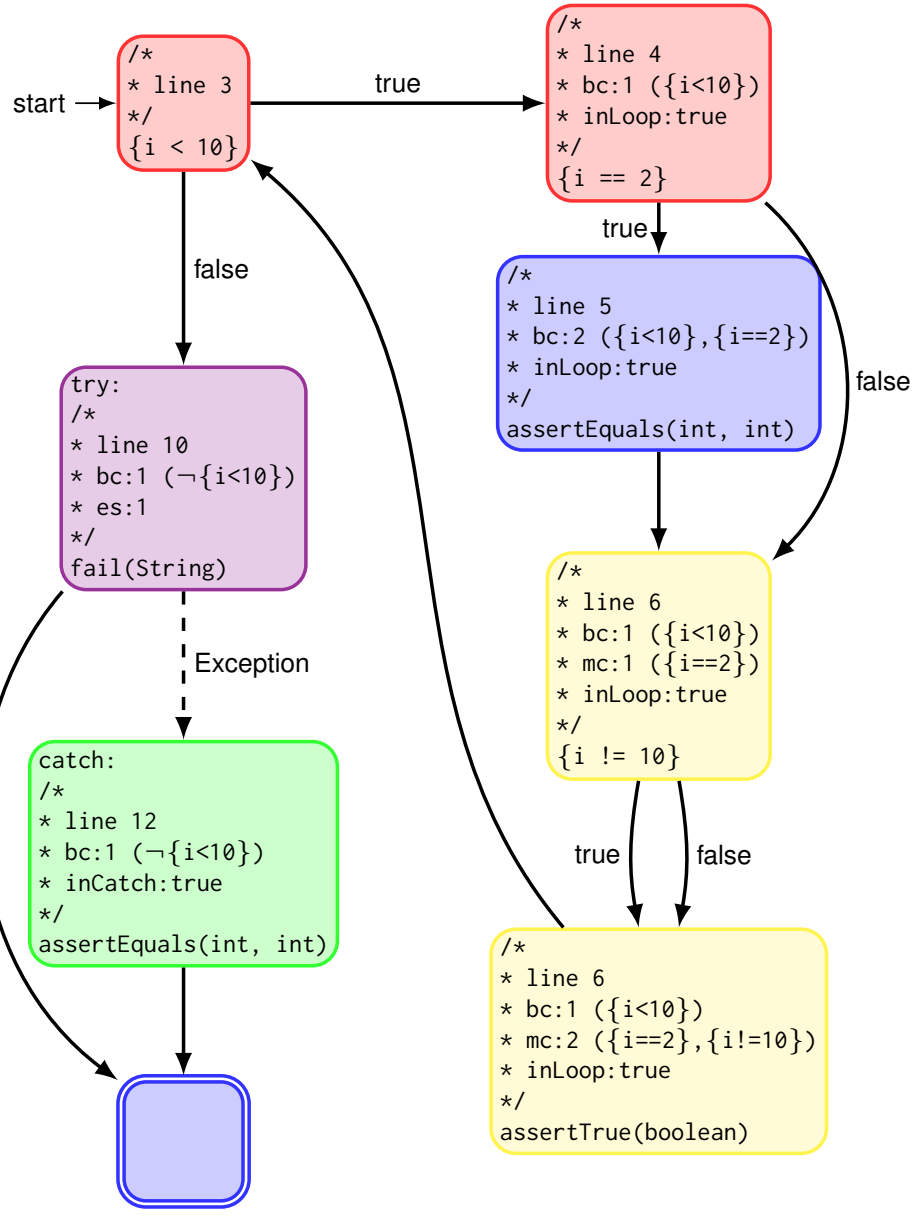
Merge Count

- Minimal number of merge vertices needed to reach vertex n from the start of the method, including n.

In-Loop Flag: Intuition

- An assertion inside a loop is likely to be different from one that is not.

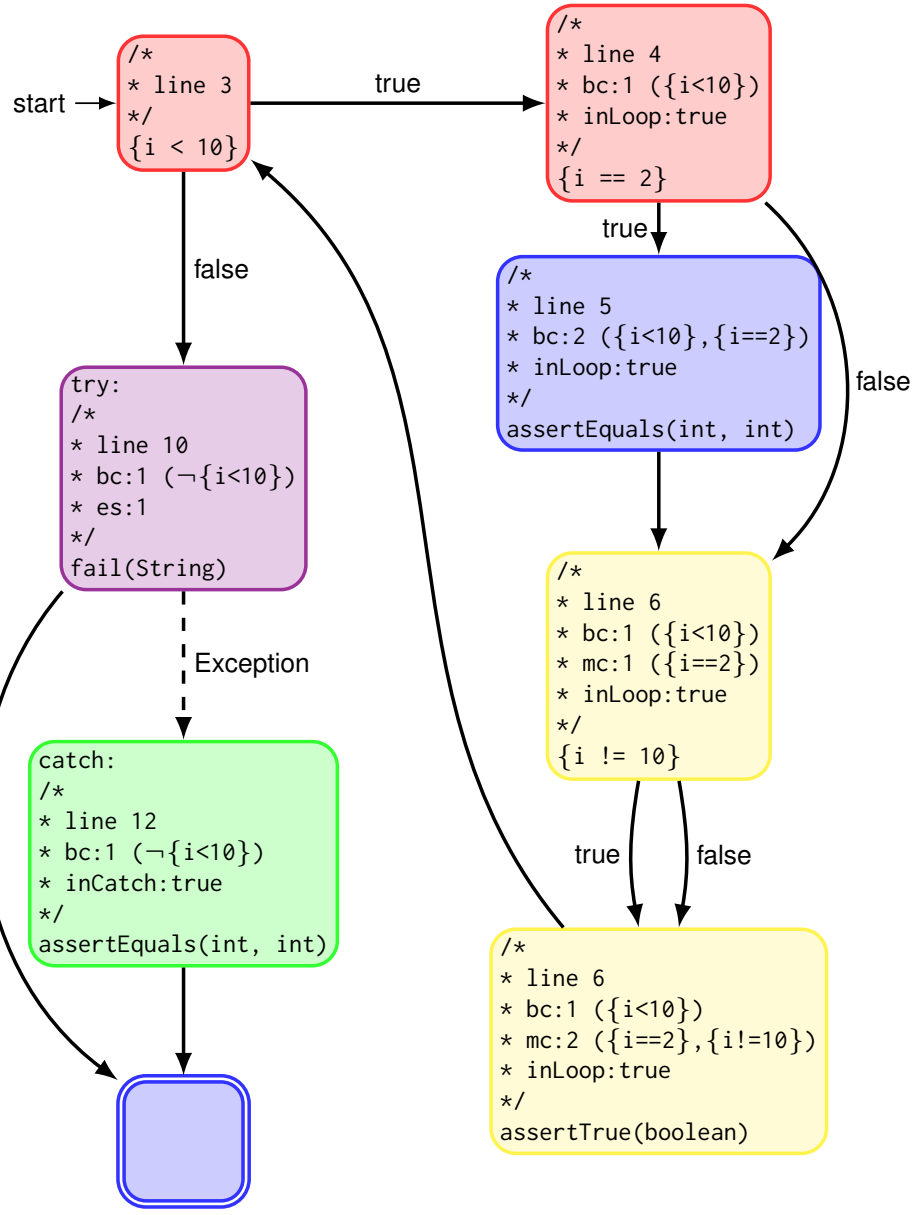
Example: CFG


start
/*
* line 3
*/
{i < 10}
true
/*
* line 4
* bc:1 ({i<10})
* inLoop:true
*/
{i == 2}
true
false
/*
* line 5
* bc:2 ({i<10},{i==2})
* inLoop:true
*/
assertEquals(int, int)
false
try:
/*
* line 10
* bc:1 (¬{i<10})
* es:1
*/
fail(String)
Exception
/*
* line 6
* bc:1 ({i<10})
* mc:1 ({i==2})
* inLoop:true
*/
{i != 10}
true
false
catch:
/*
* line 12
* bc:1 (¬{i<10})
* inCatch:true
*/
assertEquals(int, int)
/*
* line 6
* bc:1 ({i<10})
* mc:2 ({i==2},{i!=10})
* inLoop:true
*/
assertTrue(boolean)

Exceptional Successors Count: Intuition

- An assertion inside a try block with corresponding catch block(s) is likely to be different from one that is not.

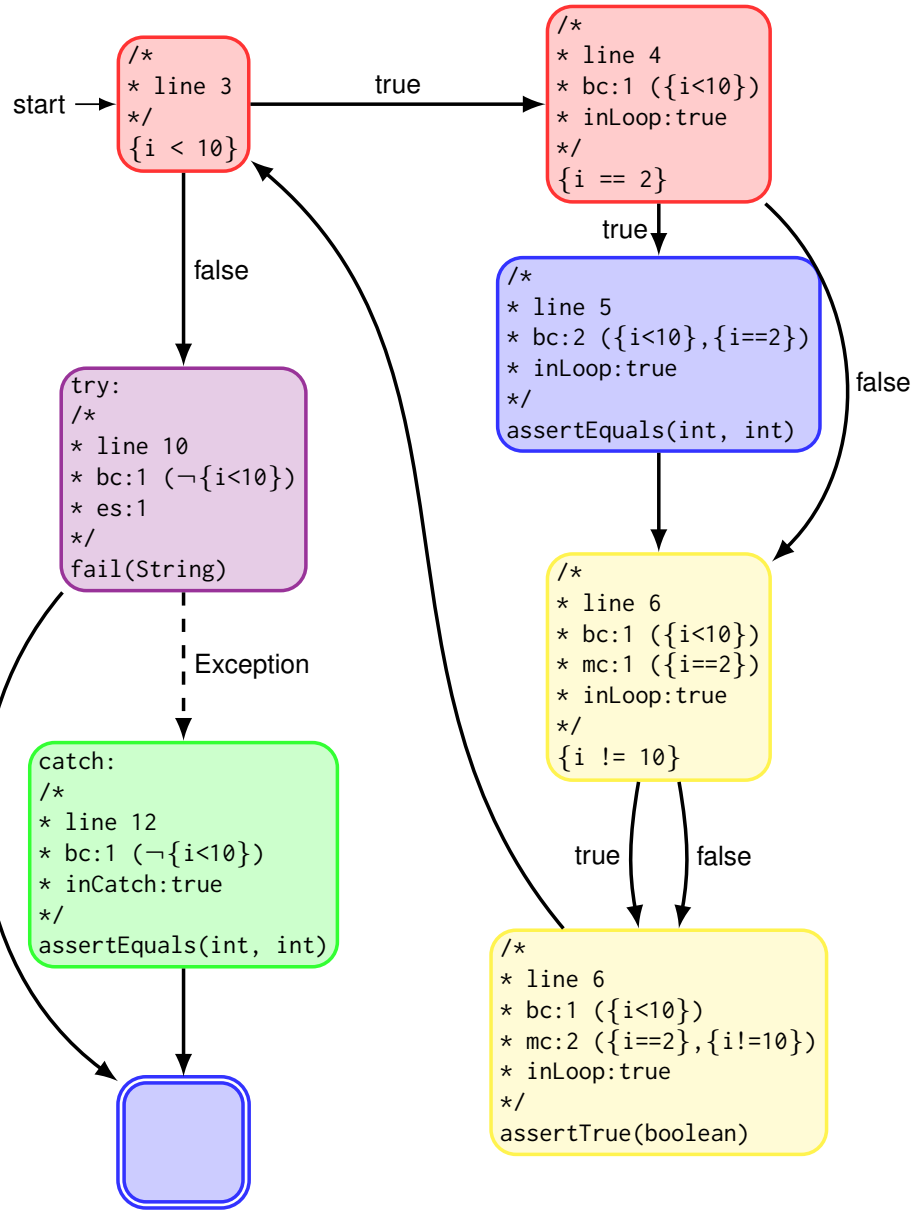
Example: CFG


start
/*
* line 3
*/
{i < 10}
true
/*
* line 4
* bc:1 ({i<10})
* inLoop:true
*/
{i == 2}
true
false
/*
* line 5
* bc:2 ({i<10},{i==2})
* inLoop:true
*/
assertEquals(int, int)
false
try:
/*
* line 10
* bc:1 (¬{i<10})
* es:1
*/
fail(String)
Exception
/*
* line 6
* bc:1 ({i<10})
* mc:1 ({i==2})
* inLoop:true
*/
{i != 10}
true
false
catch:
/*
* line 12
* bc:1 (¬{i<10})
* inCatch:true
*/
assertEquals(int, int)
/*
* line 6
* bc:1 ({i<10})
* mc:2 ({i==2},{i!=10})
* inLoop:true
*/
assertTrue(boolean)

In-Catch-Block Flag: Intuition

- An assertion inside a catch block is likely to be different from one that is not.

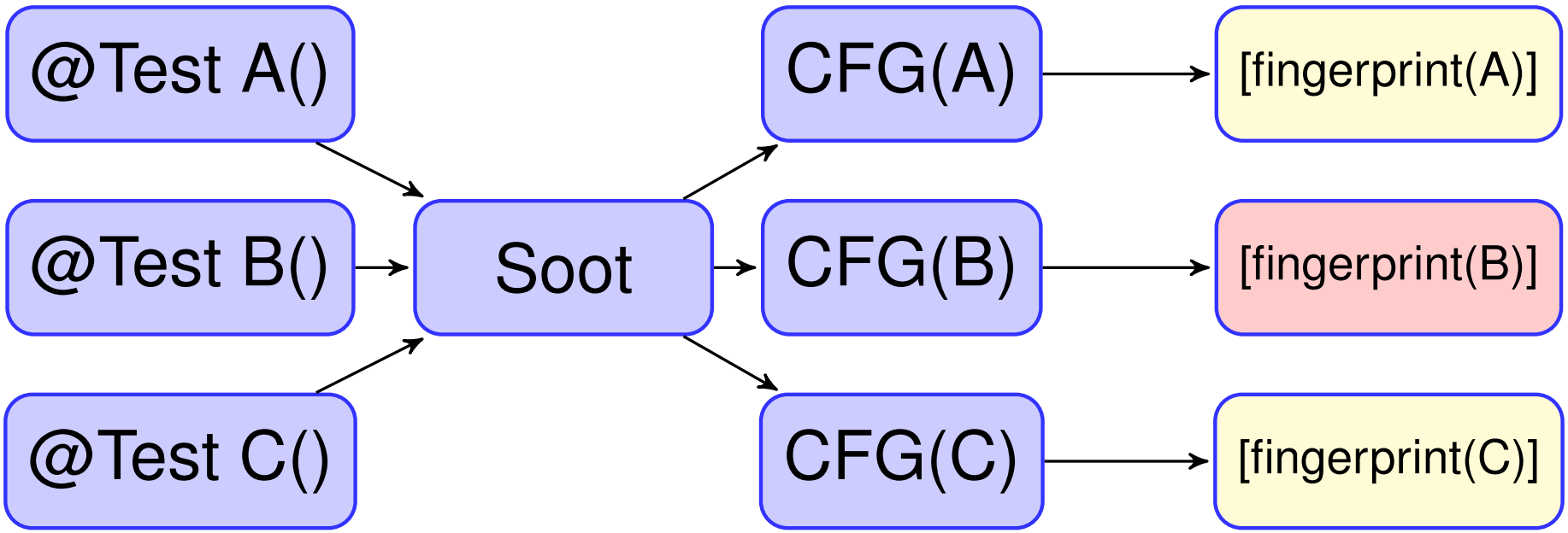
Example: CFG


start
/*
* line 3
*/
{i < 10}
true
/*
* line 4
* bc:1 ({i<10})
* inLoop:true
*/
{i == 2}
true
false
/*
* line 5
* bc:2 ({i<10},{i==2})
* inLoop:true
*/
assertEquals(int, int)
false
try:
/*
* line 10
* bc:1 (¬{i<10})
* es:1
*/
fail(String)
Exception
/*
* line 6
* bc:1 ({i<10})
* mc:1 ({i==2})
* inLoop:true
*/
{i != 10}
true
false
catch:
/*
* line 12
* bc:1 (¬{i<10})
* inCatch:true
*/
assertEquals(int, int)
/*
* line 6
* bc:1 ({i<10})
* mc:2 ({i==2},{i!=10})
* inLoop:true
*/
assertTrue(boolean)

Filtering: Must satisfy one of the following

- Contain some control flow
- Contain more than 4 assertions
- Heterogeneous in signature
  (invoke different assertion types)

Evaluation: Implementation



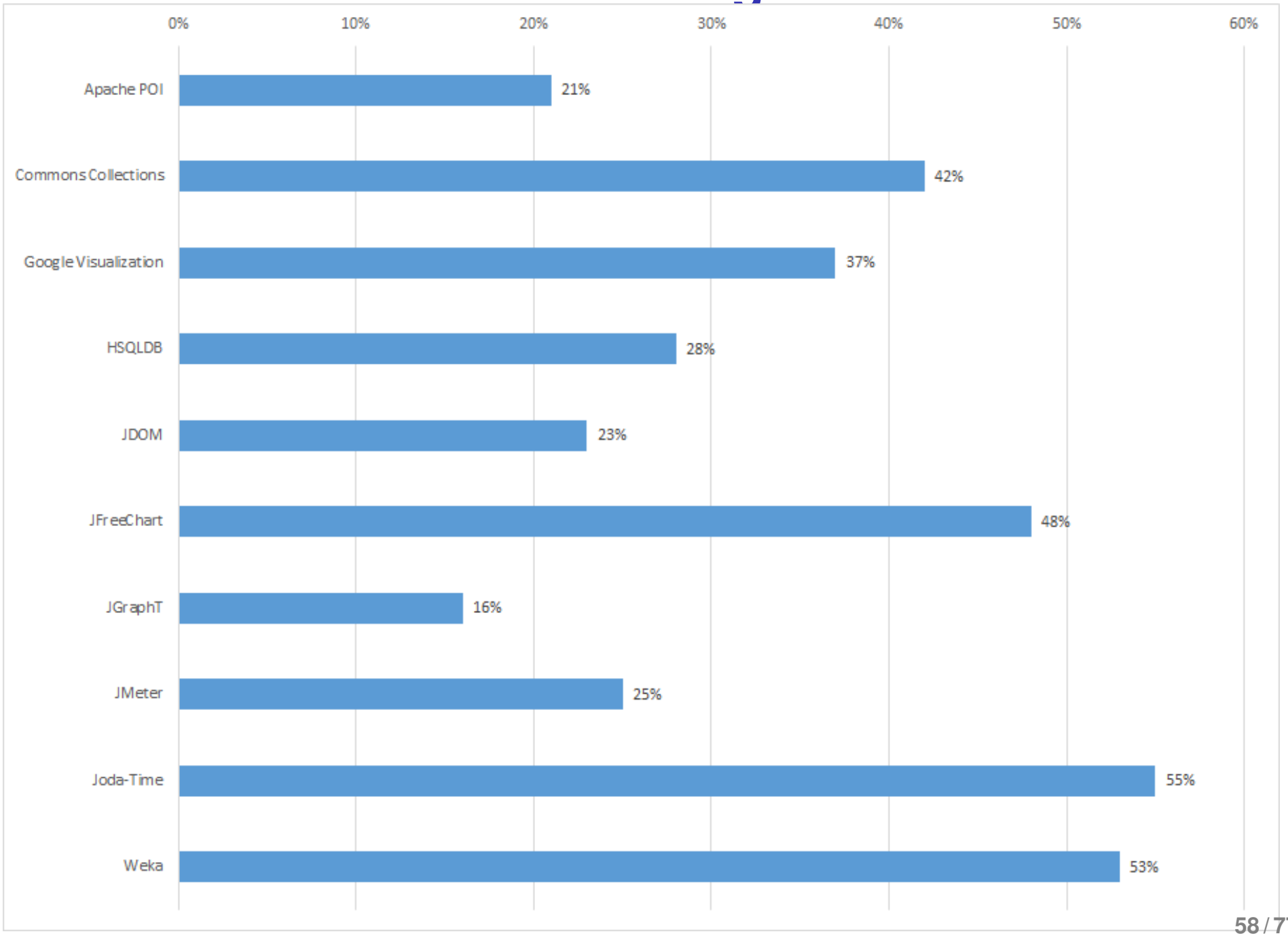
Evaluation: Benchmarks

	Test LOC	Total LOC	% Test LOC
Apache POI	86 113	247 799	35%
Commons Collections	46 129	110 394	42%
Google Visualization	13 440	31 416	43%
HSQLDB	30 481	32 208	95%
JDOM	25 618	76 734	33%
JFreeChart	93 404	317 404	29%
JGraphT	12 142	41 801	29%
JMeter	20 260	182 293	11%
Joda-Time	67 978	134 758	50%
Weka	26 270	495 198	5%

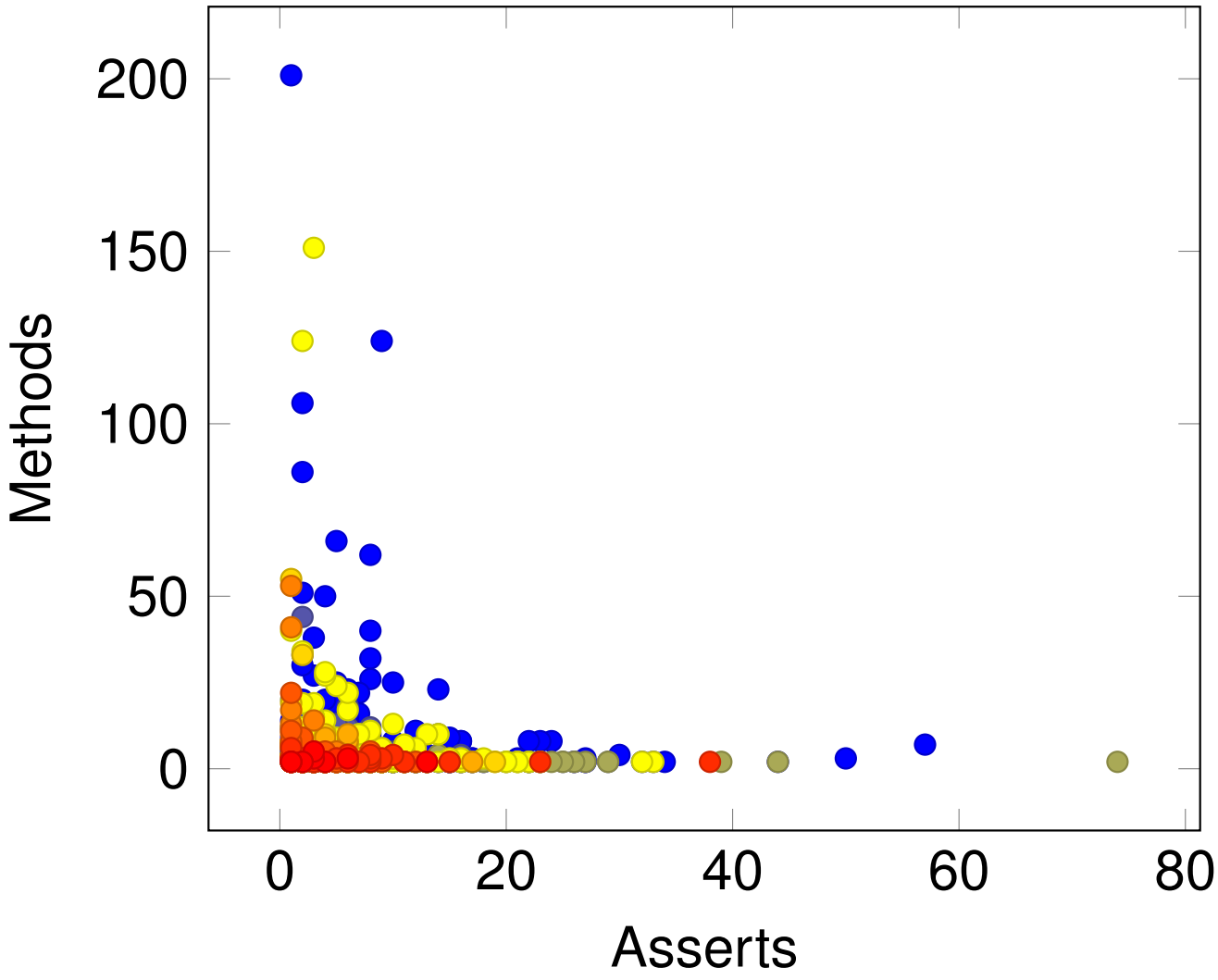
Evaluation: Analysis Run Time (seconds)

Apache POI	113
Commons Collections	50
Google Visualization	240
HSQLDB	233
JDOM	25
JFreeChart	70
JGraphT	43
JMeter	70
Joda-Time	45
Weka	91
Total	994

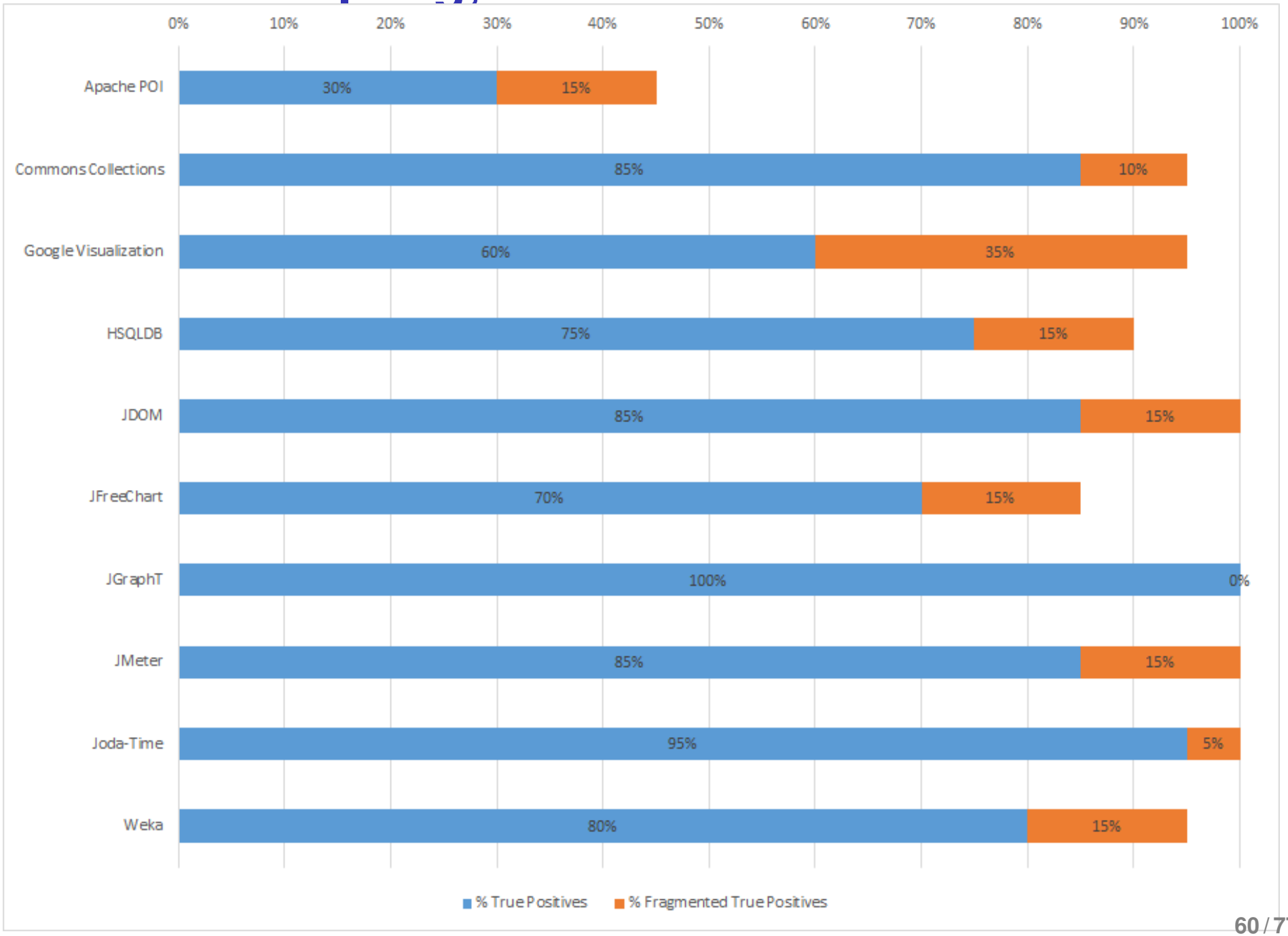
Results: Detected Refactoring Candidates



Results: Asserts vs Methods distribution



Results: Sampling, 75% True Positive rate



Qualitative Analysis: JGraphT

- Small and heterogenous tests that are unlikely to be false positives

Qualitative Analysis: Joda-Time

- Wide hierarchy of tests with identical structures and straight-line assertions.

Qualitative Analysis: Weka, Apache Commons Collections

- Textually identical clones of methods with similar data types but with different environment setups.

Qualitative Analysis: JDOM

```
public void test_TCC___String() {
  // [... 4x assertTrue(String, boolean)]
  try {
    // ...
    fail("allowed creation of an element with no name");
  } catch (IllegalNameException e) {
    // Test passed!
  }
}
```

Qualitative Analysis: Google Visualization

- Complex query-related statements and helper methods reduce the roles of assertions in a test method, resulting in a below-average true positive rate.

Qualitative Analysis: Refactorability

- Test methods that show structural similarities are most likely amenable to refatoring, however;
- Non-parametrized and small methods are diffcult to refactor.

Next Step

Guided test refactoring.

Future Perspectives

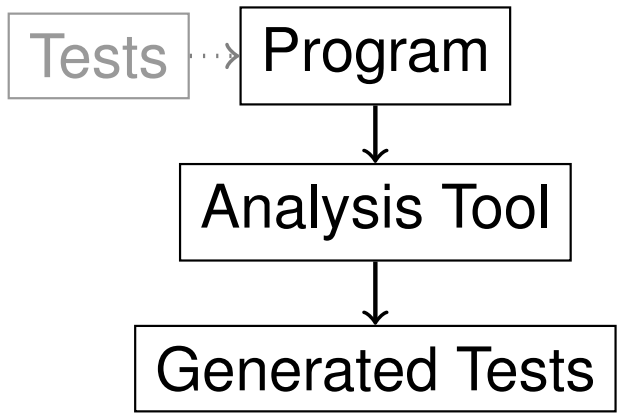
What Do We Do With Tests?

Tradtionally:

run the test, get yes/no answer.

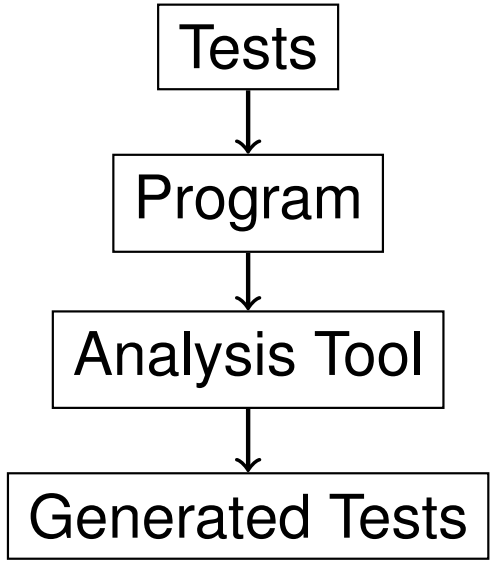
(Also, can combine with DSD/concolic analysis.)

Our usual interaction with tests (static)



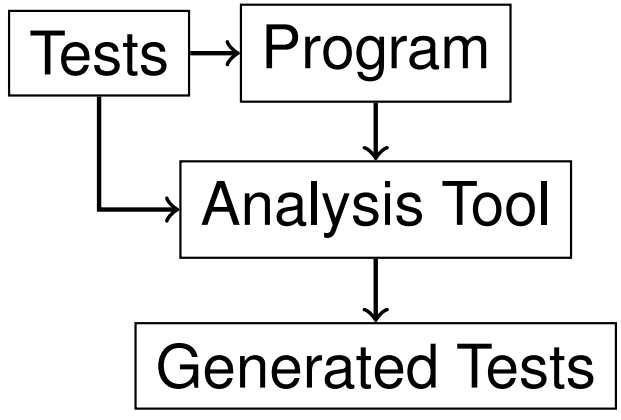
Statically, we usually just ignore tests.

Our usual interaction with tests (dynamic)



Tests are write-only with respect to the tool.

A Better Way



Why is it hard to write tests?

Need to:

1. get system under test in appropriate state;
2. decide what the right answer is.

Useful hints for static analysis!

Unit tests also illustrate. . .

- interesting points in execution space, with
- complete execution environments
  for program fragments.

Challenges

How to combine information from test runs?

What can we learn from failing tests?

Conclusions

Tests: An Opportunity for Program Analysis

We can go beyond test generation.

Tests are a valuable source of information about their associated programs.