Test Oracle Generation and Mutation Testing

Outline

- Generating Test Oracles

- Mutation Testing (Lab today)

Approaches to Generate Test Oracles

1. Specification mining techniques
   a. Creates oracles from JavaDoc comments
2. IR approach
   a. *Retrieves* a similar test and uses its assertion
3. Neural Techniques

Information Retrieval

- Information retrieval (IR) is the process of obtaining relevant information from a large collection of documents.
- Used in search engines
- Fetch the object that *best matches* a given query from a database / corpus
- Best Match is defined with a *Similarity Metric*

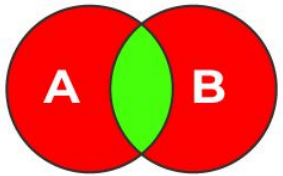
Information Retrieval for Assertion Generation

Given a Test Prefix and a corpus of Tests, find the most similar prefix and retrieve its assertion.

Similarity Metric - Jaccard

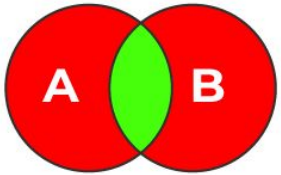
- Measures similarity between two sets
- Values range from o (no similarity) to 1 (identical sets)

$$J(X, Y) = |X \cap Y| / |X \cup Y|$$



$$\text{Jaccard} = \frac{\text{Intersection (A, B)}}{\text{Union (A, B)}}$$

Jaccard Coefficient



$$\text{Jaccard} = \frac{\text{Intersection (A, B)}}{\text{Union (A, B)}}$$

D1:

"Information Retrieval is useful"

D2:

"Retrieval of information is important"

{ information, retrieval, is useful }

{ retrieval, of, information, is, important }

$(A \cap B) = 3$

$(A \cup B) = 6$

$J(D1, D2) = 3 / 6 = 0.5$

Issues with Jaccard

1. It doesn’t consider *frequency*
2. Often times, rare words are more informative than frequent words. Jaccard doesn’t consider this

Jaccard over Test Cases

D1:

```
public void testKeyedValues() {
    KeyedValues kv;
    kv = new KeyedValues();
    Short short0 = new Short(2);
    kv.insertValue(0, short0, 2);
    kv.removeValue(0);
}
```

```
{public, void, testKeyedValues,
KeyedValues, kv, new, Short,
short0, insertValue, removeValue}
```

D2:

```
public void testMultipleKeyedValues() {
    KeyedValues kv = new KeyedValues();
    kv.insertValue(1, "First", 10);
    kv.insertValue(2, "Second", 20);

    int value2 = kv.getValue(2);

    assertEquals(20, value2);
}
```

```
{public, void, testMultipleKeyedValues,
KeyedValues, kv, insertValue, getValue,
assertEquals}
```

Jaccard over Test Cases

D1:

```
{public, void,
testKeyedValues,
KeyedValues, kv, new,
Short, short0, insertValue,
removeValue}
```

D2:

```
{public, void,
testMultipleKeyedValues,
KeyedValues, kv, insertValue,
getValue, assertEquals}
```

$(A \cap B) = 5$
$(A \cup B) = 12$

$J(D1, D2) = 5 / 12 = .4167$

IR with Jaccard

Given a test case and a corpus of tests, retrieve a test from the corpus with the highest jaccard similarity and inspect the assertion

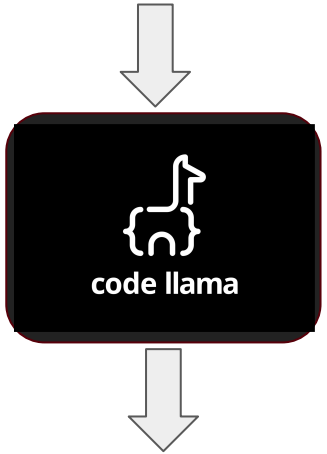
```
public void testKeyedValues() {
    KeyedValues kv;
    kv = new KeyedValues();
    Short short0 = new Short(2);
    kv.insertValue(0, short0, 2);
    kv.removeValue(0);
}
```

```
public void testRemove() {
    KeyedValues values;
    values = new KeyedValues();
    values.insertValue(0, 7, 2);
    values.removeValue(0);

   assertEquals(values.size(), 0);
}
```

IR with Jaccard - small neural edit to the retrieved assertion

```
public void testKeyedValues() {
    KeyedValues kv;
    kv = new KeyedValues();
    Short short0 = new Short(2);
    kv.insertValue(0, short0, 2);
    kv.removeValue(0);
}
```

assertEquals(values.size(), 0);



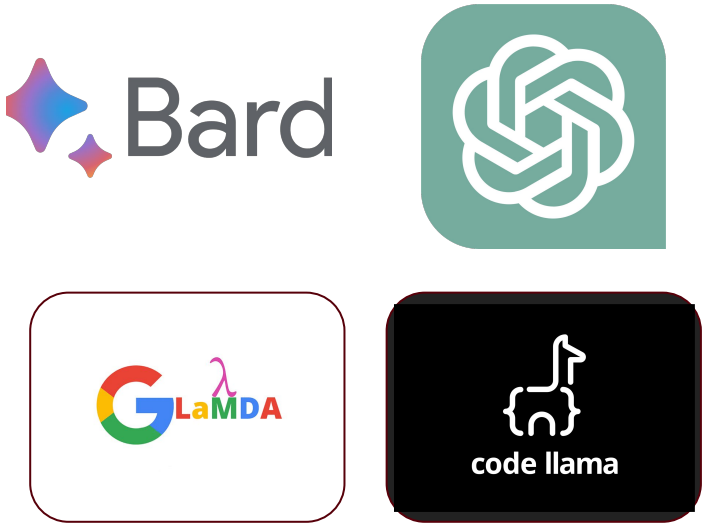
assertEquals(kv.size(), 0);

Approaches to Generate Test Oracles

1. Specification mining techniques
   a. Creates oracles from JavaDoc comments
2. IR approach
   a. *Retrieves* a similar test and uses its assertion
3. **Neural Techniques**

Neural Oracle Generation

Given a **test** and **focal method**, generate an assertion token by token



Neural Oracle Generation

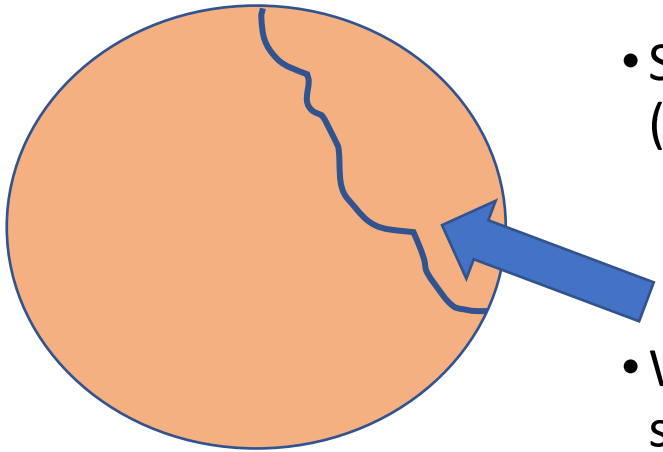
```
public void testPop() {
  Stack<int> s = new Stack<int>();
  int a = 2;

  s.push(a);
  s.pop();

  bool empty = s.isEmpty();
  <AssertPlaceHolder>
}
```

```
public void pop () {
  // NO-OP
}
```

assertTrue(s.length == 0);

NO GUARANTEES THAT THE MODEL OUTPUT COMPILES, TYPE CHECKS, OR IS CORRECT IN ANY WAY!

Leveraging Observed Oracle Structures



Set of all possible oracles

- Set of all possible oracles is massive… (maybe even infinite)
- We observed a small common set of oracle structures **in tests**

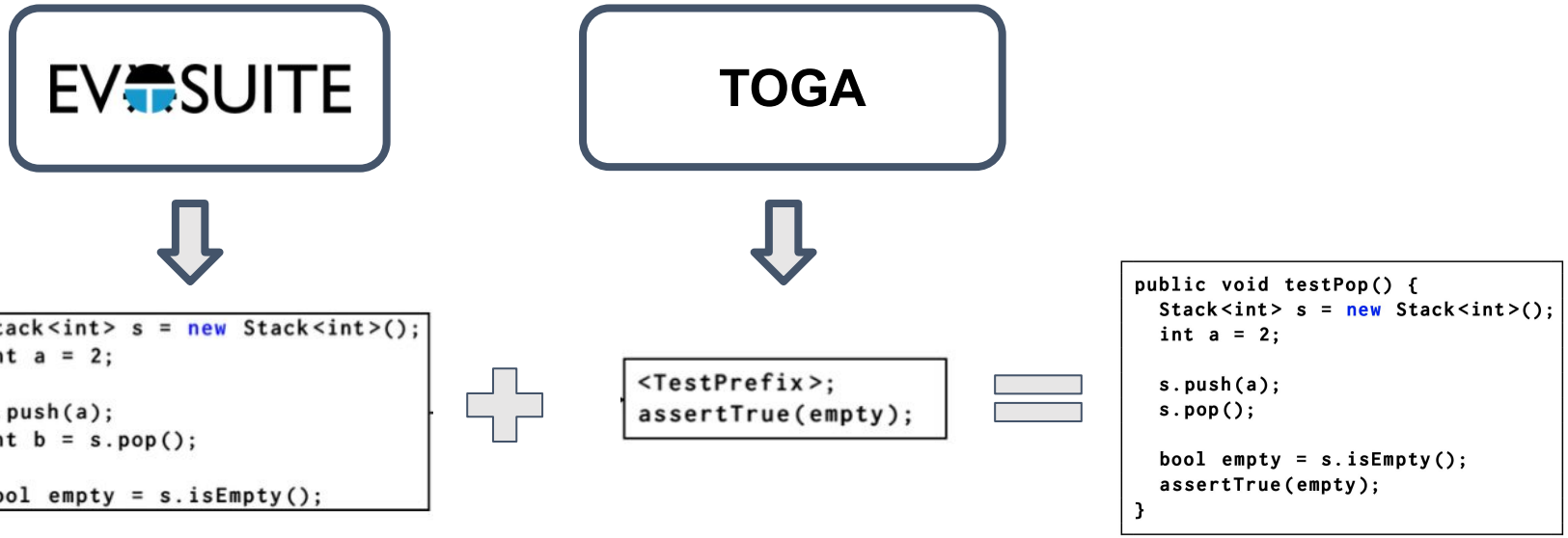
Grammatical Structure of Oracles

```
Test              T    := O(P)
Prefix            P    := statement | P; P
Oracle         O(P)    := E(P) | R(P)
Except Oracle  E(P)    := try{P; fail();} catch(Exception e){}
Return Oracle  R(P)    := P; A
Assertion         A    := assertEquals(const|var,expr) |
                          assertTrue(expr) | assertFalse(expr) |
                          assertNull(expr) | assertNotNull(expr)
```

Deliberately Restricted, but 82% tests fit grammar when evaluated on ATLAS corpus

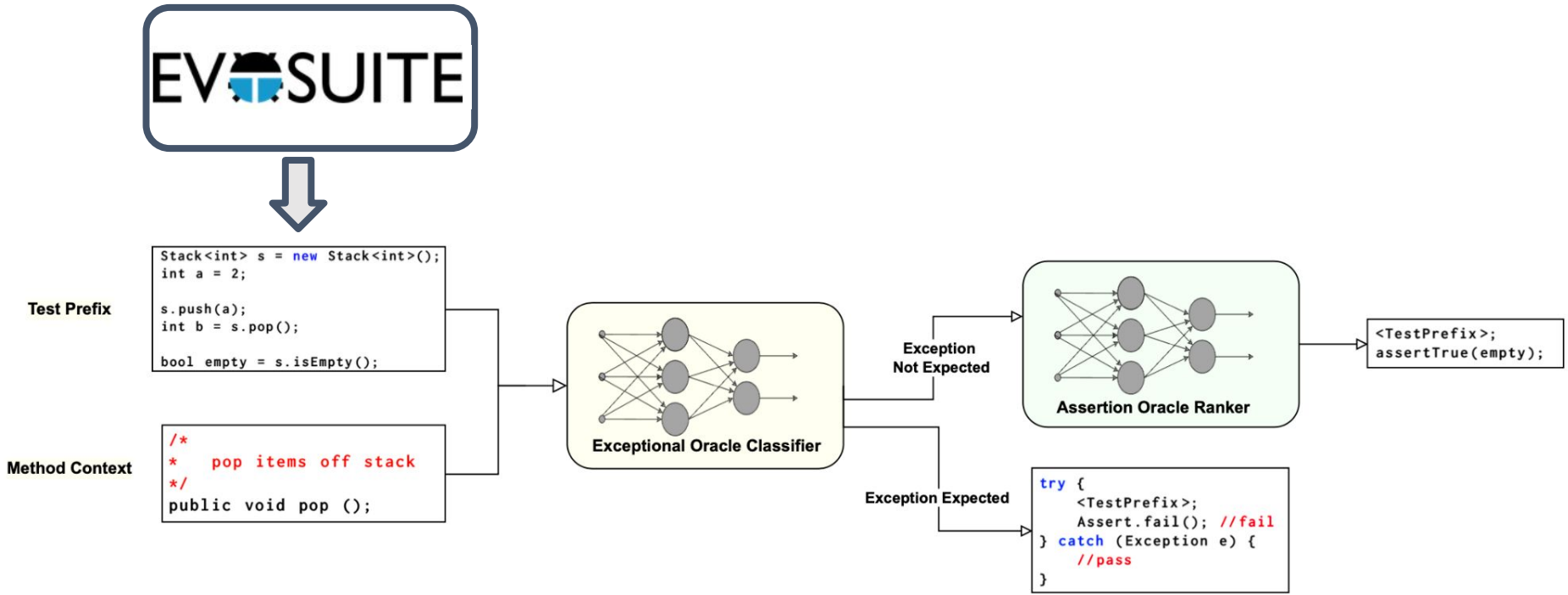
System Overview

generated by automated testing tool



System Overview

generated by automated testing tool



Assertion Inference

Method Docstring +

```
/*
*    pop items off stack
*/
public void pop ();
```

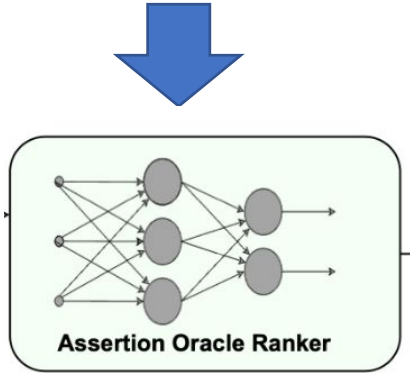
+

Assertion

```
assertEquals(2, b)
assertEquals(a, b)

assertEquals(1, b)
assertEquals(0, b)
assertEquals(100, b)
```

Test Prefix

```
public void testPop() {
  Stack<int> s = new Stack<int>();
  int a = 2;

  s.push(a);
  int b = s.pop();
  <AssertPlaceHolder
  >
}
```




```
assertEquals(2, b)
```

Evaluating Test Oracles

A Note on Scientific Evaluations

Benchmark:

Verb: running an algorithm, in order to assess the relative performance, against a number of "standard tests and trials"

Noun: the dataset which we evaluate our algorithm on. Should contain "gold standard" answers

Baseline:

The "standard tests and trials"

The algorithms we are comparing our performance against

Test Environment:

"This evaluation was conducted on a Linux machine with Intel(R) Xeon(R) E5-2690 v3 CPU (2.60GHz) and 112GB main memory."

Evaluation Metrics

1. Correctness
   a. Does the assertion generated match the assertion the developer wrote?
   b. Does the generated assertion compile?

2. Bug Finding
   a. If the implementation of the focal method was incorrect, would the assertion catch it?

3. Time taken / resources used / computational complexity

Quis custodiet ipsos custodes?

Rank the following assertions from best to worst

```
double calculatePrice(int basePrice, int daysUntilEvent, int customerAge, boolean isMember) {
        double maxDiscount = basePrice - 100;

        double discountRate = daysUntilEvent / maxDiscount;

        if (isMember) discountRate += .10;

        if (customerAge >= 65 || customerAge < 18) {
            discountRate += (customerAge - 65) / maxDiscount;
        }

        return basePrice - (basePrice * discountRate);
}
```

```
int price = BuyTicket.calculatePrice(150, 7, 28, true);

Assertion 1: assertTrue(true);
Assertion 2: assertTrue(price > 0);
Assertion 3: assertEquals(price, 36.0);
```

Quis custodiet ipsos custodes?

What makes a good assertion?

If the code changes and no longer functions as intended, the assertion should fail

Mutation Testing: modifying a program in small ways and measuring if the test suite catches the difference in behavior

Mutation Testing

Given a test suite T and a source program P

`M = genMutants(P)` //generate a set of mutants

for each mutant `m` in `M`:

    execute `T(m)` and record if passed or failed

Generating Mutants

- Make a small change to P
- Ideally, mutants should:
  - Compile
  - Model real world human defects that cause bugs
  - Be diverse

Approaches to Generate Mutants

1. Rule based
    a. Transformation rules similar to what you implemented in HW1
    b. Guarantees compilation of mutants

2. Neural
    a. NO GUARANTEES
    b. Can better model real world human defects that cause bugs
    c. Costs...

Pitest implements the following mutators

1. Conditional Boundary Mutator
2. Increments Mutator
3. Invert Negatives Mutator
4. Math Mutator
5. Negative Conditionals Mutator
6. Return Values Mutator
7. Void Method Call Mutator
8. Empty Returns Mutator
9. False Returns Mutator
10. Null Returns Mutator
11. Primitive Returns Mutator

... and more optional mutators

Conditionals Boundary Mutator (CONDITIONALS_BOUNDARY)

Active by default

The conditionals boundary mutator replaces the relational operators `<, <=, >, >=`

with their boundary counterpart as per the table below.

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

For example

```
if (a < b) {
  // do something
}
```

will be mutated to

```
if (a <= b) {
  // do something
}
```

Increments Mutator (INCREMENTS)

Active by default

The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.

For example

```
public int method(int i) {
  i++;
  return i;
}
```

will be mutated to

```
public int method(int i) {
  i--;
  return i;
}
```

Please note that the increments mutator will be applied to increments of local variables only. Increments and decrements of member variables will be covered by the Math Mutator.

Mutation Score

```
mutation score = killed mutants / total mutants
```

The proportion of the killed mutants over the entire set of mutants forms a test adequacy metric that is called **mutation score**.

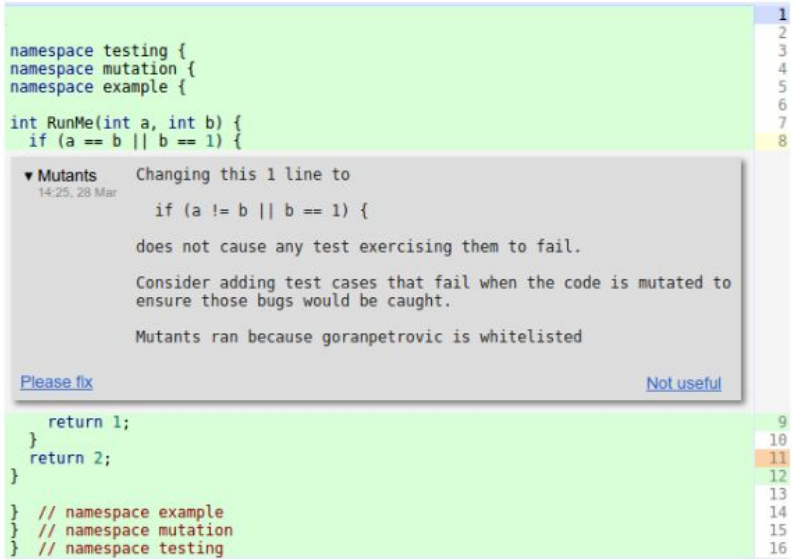


Figure 1: Mutant finding shown in the Critique - Google code review tool

Summary

- Approaches to generate oracles
- Mutation Testing
- HW1 (due Wednesday Feb 12)
  - AST and Bytecode transformations
- Lab today: Running a mutation testing tool