COMP5111 – Fundamentals of Software Testing and Analysis

Search-based Test Generation - EvoSuite

Shing-Chi Cheung

Computer Science & Engineering

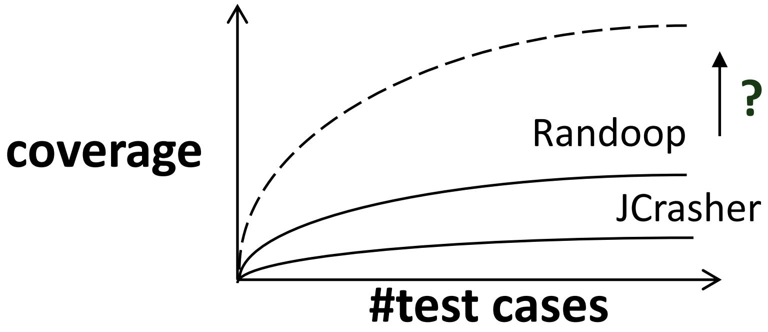
HKUST

Adapted from the presentation slides by Yongbae Park for Andrea Arcuri, Gordon Fraser, Juan Pablo Galeotti, Automated Unit Test Generation for Classes with Environment Dependencies, ASE 2014

Limitations of Randoop

Coverage saturates quickly with increasing amount of test cases

- Generates new test cases randomly
  - Quality of new test cases is not guaranteed
- Weak test oracles
  - Five built-in rules
- Not working when handling environment APIs
  - Date, SystemInUtil, InputStream



Limitations of Randoop

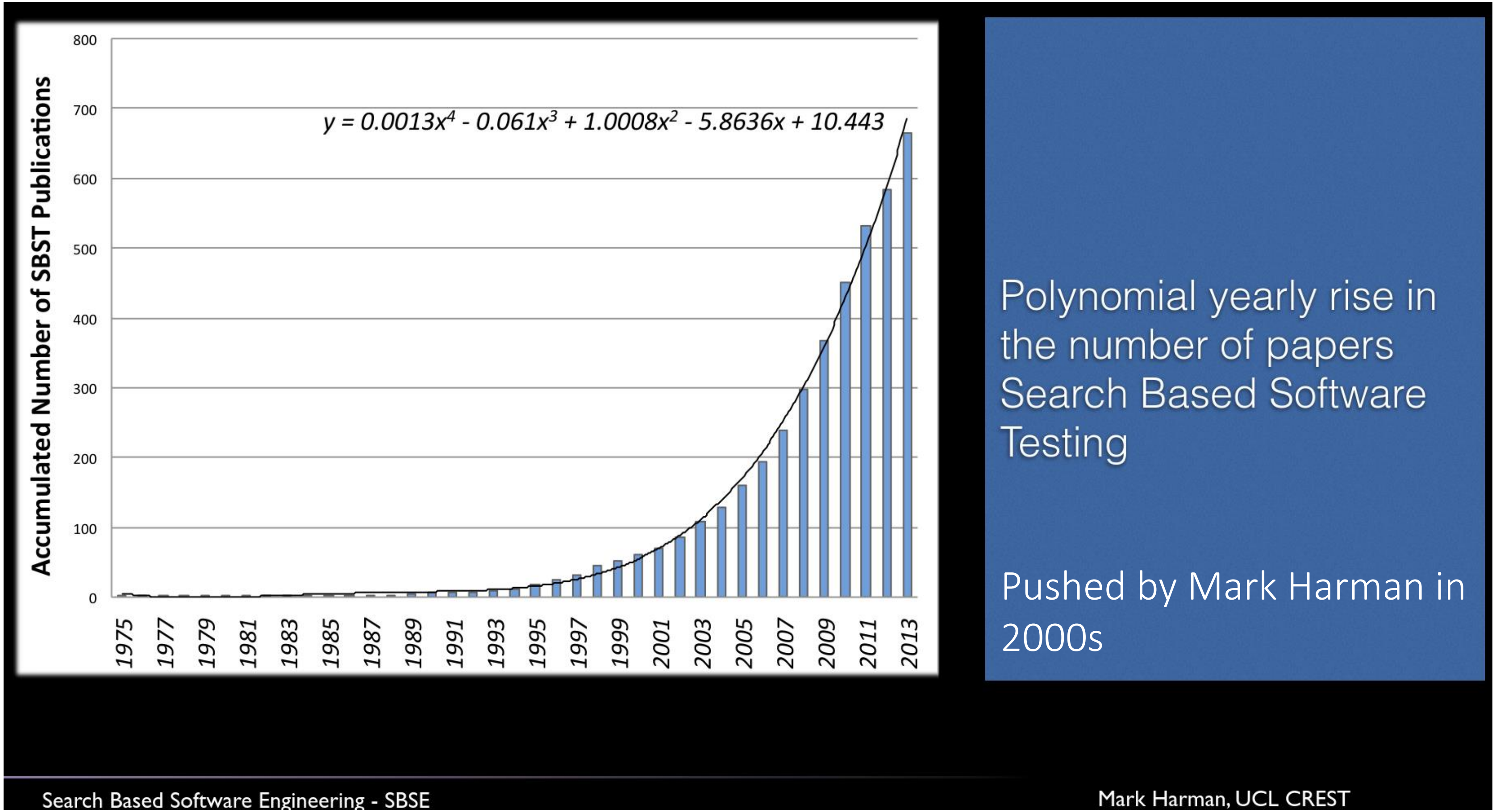
Coverage saturates quickly with increasing amount of test cases

- Generates new test cases randomly
  - Quality of new test cases is not guaranteed
- Weak test oracle
  - Five built-in rules
- Not working when the code being tested calls environment APIs
  - Date, SystemInUtil, InputStream

Coverage-driven

Regression & Mutation-based oracle

API mocking



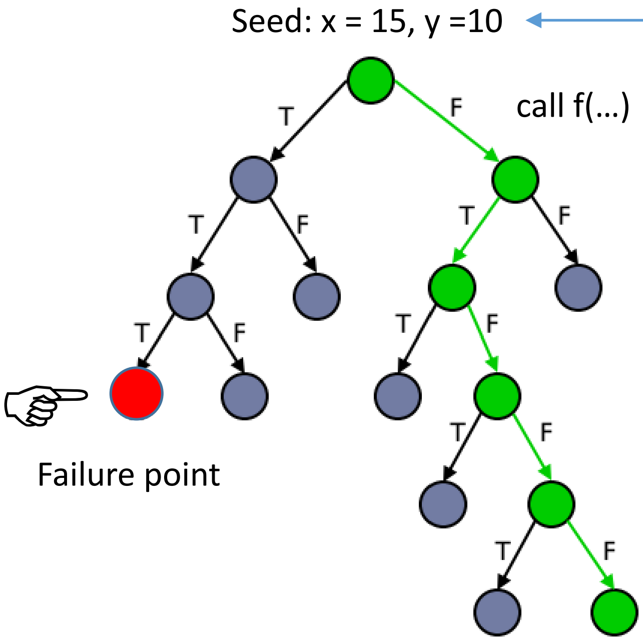
Polynomial yearly rise in the number of papers Search Based Software Testing

Pushed by Mark Harman in 2000s

Search Based Software Engineering - SBSE

Mark Harman, UCL CREST

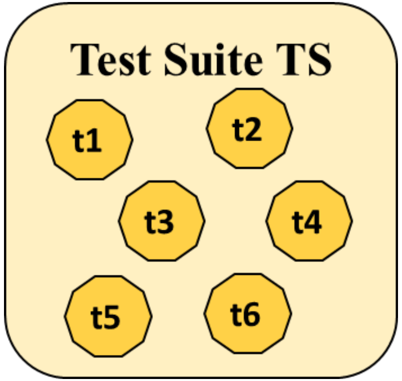
Motivation: Challenge of Concolic Testing



Find diversified concrete test executions (seeds) to visit a path close to the triggering of fault

Search Goal – EvoSuite

- Given a class under test (CUT), EvoSuite automatically generates a test suite (TS) using a genetic algorithm
- TS achieves high code coverage
- Supports mutation analysis to generate test oracles (i.e., assertions).
- Supports dynamic symbolic execution (a.k.a. concolic testing) to reach difficult branches
- http://www.evosuite.org/



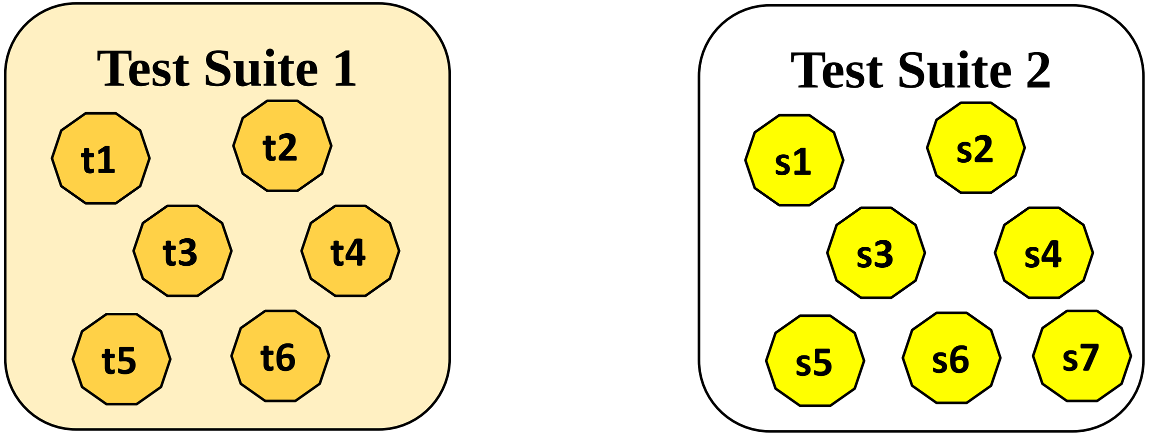
One Generation Strategy – EvoSuite

Given a class under test (CUT), EvoSuite automatically generates a test suite using a genetic algorithm:

1. Create 2 initial test suites by calling methods randomly; insert them into current generation.
2. Select two test suites from current generation.
3. Create 2 new test suites by crossover (exchange test cases of the suites).
4. Modify two test suites from step 3 with mutation operators (insert, remove, change operators).
5. Insert the new two test suites from step 4 to next generation if coverage of the new two test suites are higher than that of their parents.
6. Repeat 2~5 until there are enough test suites for the next generation
7. Repeat 2~6 until time limit is reached or all branches (or specified coverage) are covered
8. Select a test suit with the highest branch coverage (or specified coverage) and insert assertions by executing the tests

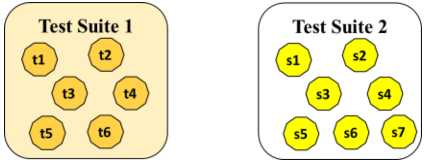
EvoSuite

1. Create 2 initial test suites by adding method calls randomly and insert the test suites into current generation.

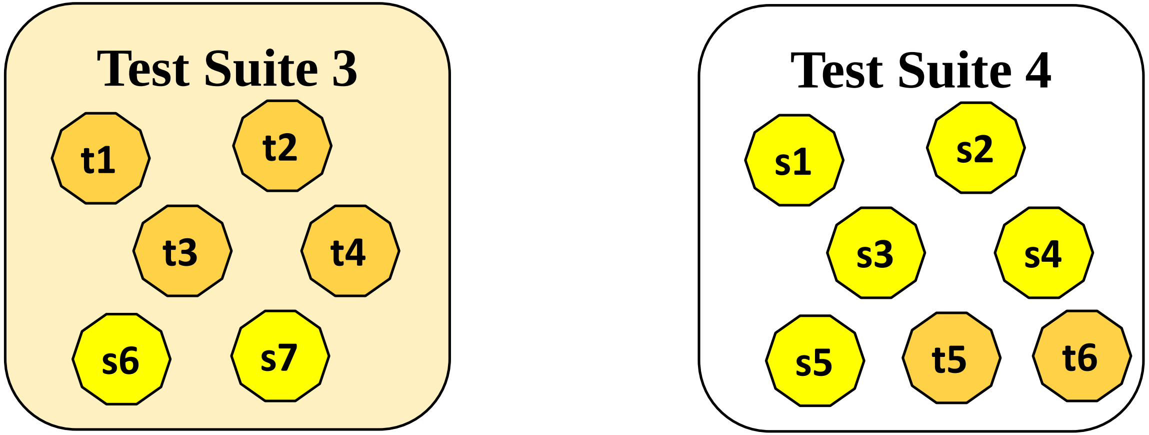


EvoSuite

2. Select two test suites from current generation



3. Create 2 new test suites by crossover (exchange test cases of the suites)



Example for EvoSuite

```java
public class Message {
  private Date created;
  private String text;
  public Message(String text) {
    this.created=new Date();
    this.text=(text==null?””:text);
  }
  public String toString() {
    return created+”,”+text;
  }
  public String getText() {
    return text;
} }
```

- `Message` class contains 2 members:
  - `created` holds the message creation date, which is set in the constructor of `Message`
  - `text` holds the contents of a message

Initial Test Suite Generation

- EvoSuite randomly inserts a small number of new statements for the empty test cases of the initial test suites

```
public class Message {
  private Date created;
  private String text;
  public Message(String text) {
    this.created=new Date();
    this.text=(text==null?””:text);
  }
  public String toString() {
    return created+”,”+text;
  }
  public String getText() {
    return text;
} }
```

```
public class TestSuite1 {
  public void test0() { //length 3
    Message v0 = new Message(“e”);
    Message v1 = new Message(“c”);

}
  public void test1() { //length 2
    Message v0 = new Message(null);

} }
```

Initial Test Suite Generation

- In each step, EvoSuite adds a method call whose callee is a method of a <u>class under test</u> or a method call of <u>an object that is available at the end</u>
  - A parameter of the created method call is selected from available values, `null`, or a random value

Class under test

```
public class Message {
  private Date created;
  private String text;
  public Message(String text) {
    this.created=new Date();
    this.text=(text==null?””:text);
  }
  public String toString() {
    return created+”,”+text;
  }
  public String getText() {
    return text;
} }
```

```
public class TestSuite1 {
  public void test0() { //length 3
    Message v0 = new Message(“e”);
    Message v1 = new Message(“c”);
    String v2 = v1.toString();
}
  public void test1() { //length 2
    Message v0 = new Message(null);
    String v1 = v0.getText();
} }
```

Initial Test Suite Generation

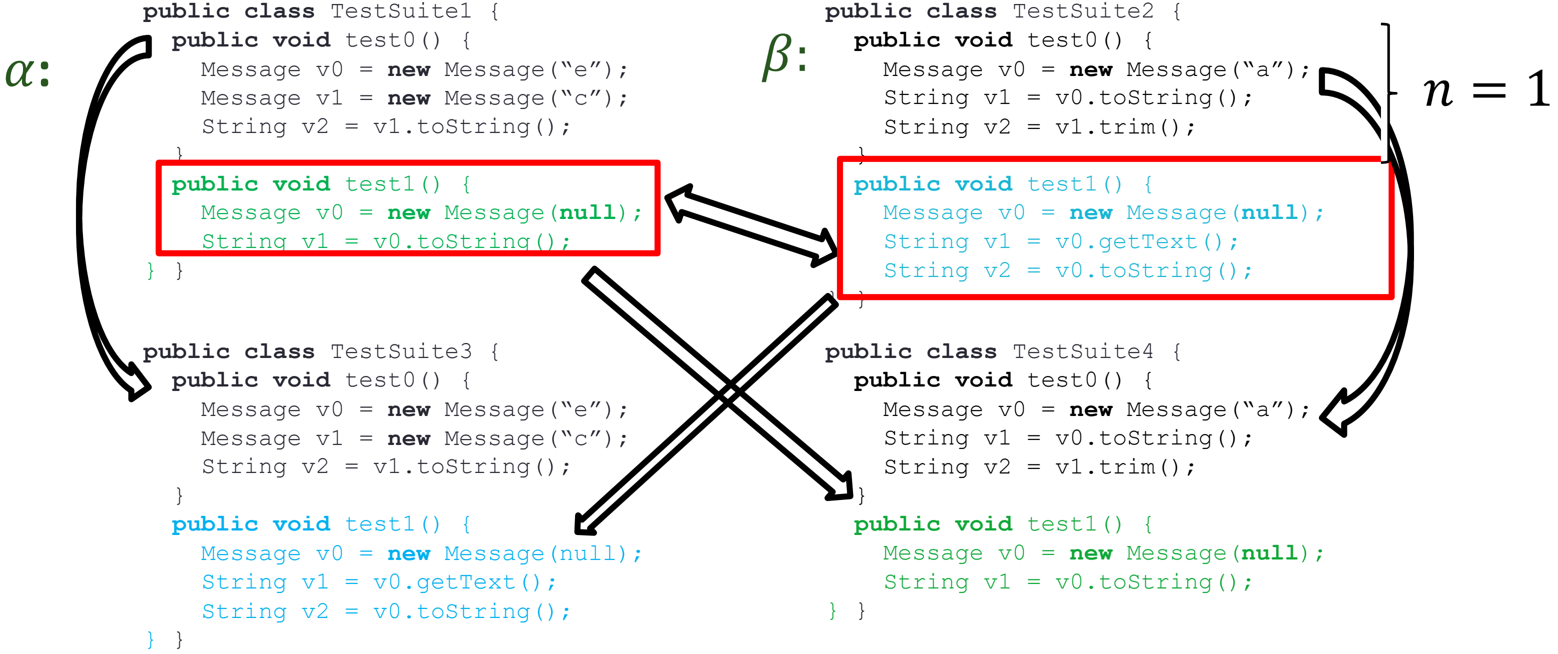
- The maximum length for each test case is set by a random number (usually a small integer).

```
public class Message {
  private Date created;
  private String text;
  public Message(String text) {
    this.created=new Date();
    this.text=(text==null?””:text);
  }
  public String toString() {
    return created+”,”+text;
  }
  public String getText() {
    return text;
} }
```

```
public class TestSuite1 {
  public void test0() { //length 3
    Message v0 = new Message(“e”);
    Message v1 = new Message(“c”);
    String v2 = v1.toString();
  }
  public void test1() { //length 2
    Message v0 = new Message(null);
    String v1 = v0.getText();
  }
  …
}
```

Crossover *Random or Two with the highest coverage*

- EvoSuite selects test suites $\underbrace{\alpha \text{ and } \beta}$ from current population, and swaps the last n test cases of α with the last n test cases of β, where n is a random number



α:

```
public class TestSuite1 {
  public void test0() {
    Message v0 = new Message(“e”);
    Message v1 = new Message(“c”);
    String v2 = v1.toString();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
} }
```

β:

```
public class TestSuite2 {
  public void test0() {
    Message v0 = new Message(“a”);
    String v1 = v0.toString();
    String v2 = v1.trim();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.getText();
    String v2 = v0.toString();
  }
```

$n = 1$

```
public class TestSuite3 {
  public void test0() {
    Message v0 = new Message(“e”);
    Message v1 = new Message(“c”);
    String v2 = v1.toString();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.getText();
    String v2 = v0.toString();
} }
```

```
public class TestSuite4 {
  public void test0() {
    Message v0 = new Message(“a”);
    String v1 = v0.toString();
    String v2 = v1.trim();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
} }
```

Crossover

- EvoSuite selects test suites $\overbrace{\alpha \text{ and } \beta}^{\textbf{\textit{Random or Two with the highest coverage}}}$ from current population, and swaps the last n test cases of α with the last n test cases of β, where n is a random number

α:

```java
public class TestSuite1 {
  public void test0() {
    Message v0 = new Message(“e”);
    Message v1 = new Message(“c”);
    String v2 = v1.toString();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
} }
```

β:

```java
public class TestSuite2 {
  public void test0() {
    Message v0 = new Message(“a”);
    String v1 = v0.toString();
    String v2 = v1.trim();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.getText();
    String v2 = v0.toString();
} }
```

$n = 1$

Two offsprings

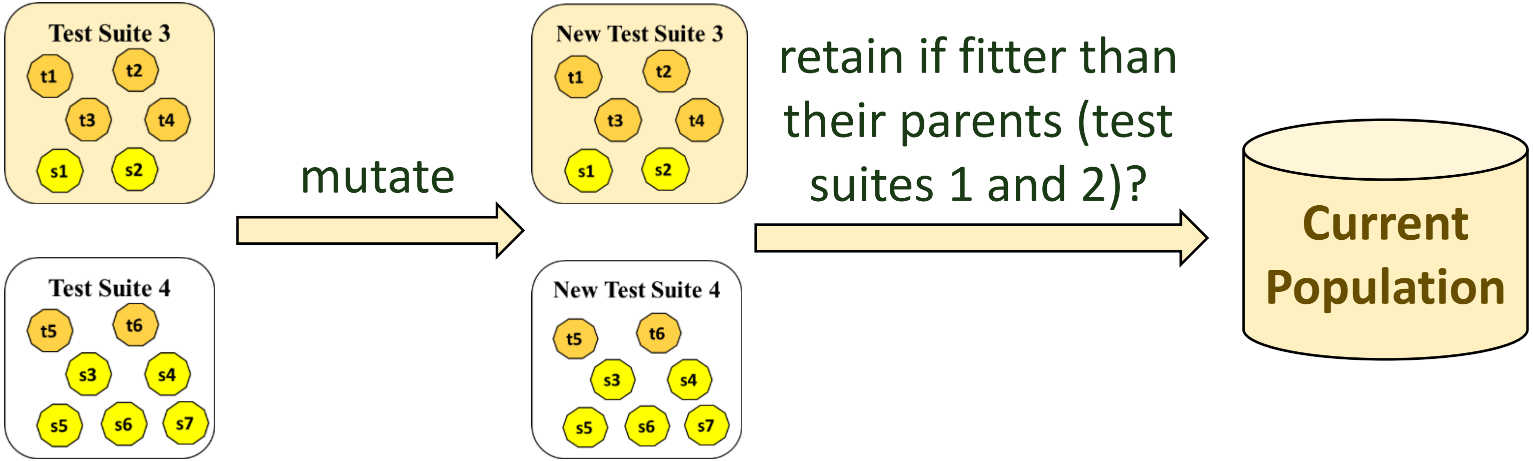
```java
public class TestSuite3 {
  public void test0() {
    Message v0 = new Message(“e”);
    Message v1 = new Message(“c”);
    String v2 = v1.toString();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.getText();
    String v2 = v0.toString();
} }
```

```java
public class TestSuite4 {
  public void test0() {
    Message v0 = new Message(“a”);
    String v1 = v0.toString();
    String v2 = v1.trim();
  }
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
} }
```

Mutate offsprings with insert, remove and change operators

EvoSuite

4. Modify the two test suites from step 3 with mutation operators (insert, remove, change operators).
5. Insert the new two test suites from step 4 to next generation if coverage of the new two test suites are higher than that of their parents.



Mutation – Insert Operator

- Insert operator adds a new statement at a random position using one of the two ways

1. Add a method call statement whose callee is a method of the class under test
2. Add a method call of an object that is available at a random position
   - ❑ A parameter of the created method call is selected from available values, `null`, or a random value

```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
} }
```

Adding a method call of a class under test

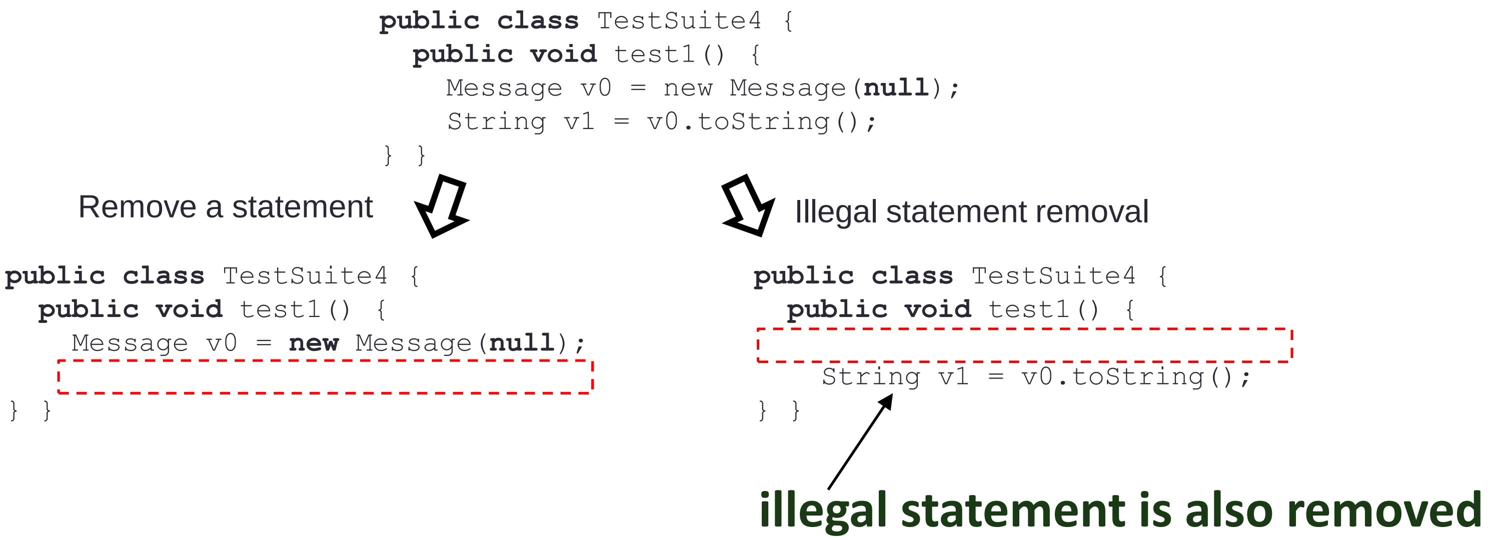
```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
    Message v2 = new Message("b");
} }
```

Adding a method call of an available object available at a random position

```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message("a");
    boolean v2 = v0.equals(null);
    String v1 = v0.toString();
} }
```

Mutation – Remove Operator

- Randomly selects a statement in a test case and remove the statement



```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
} }
```

Remove a statement

```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message(null);

} }
```

Illegal statement removal

```
public class TestSuite4 {
  public void test1() {

    String v1 = v0.toString();
} }
```

illegal statement is also removed

Mutation – Change Operator

- Change operator randomly changes a callee method or a parameter of a method invocation statement in a test case
  - Selects a new callee method that whose return type is same as the original method.
  - Changes an argument of a method call into a value which is available from the previous statements or a random value.

Do you find any assumption made by this change operator?

```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
} }
```

callee method change ⇩

```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message(null);
    String v1 = v0.getText();
} }
```

⇩ parameter change (random value)

```
public class TestSuite4 {
  public void test1() {
    Message v0 = new Message(“a”);
    String v1 = v0.toString();
} }
```

Add test suites to current generation

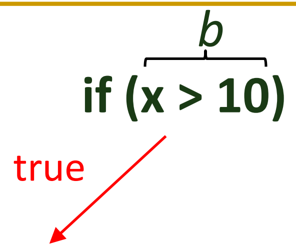
- Check the coverage of new Test Suites 3 and 4 after mutation.
- Add a Test Suite if it is fitter than its parents.
  // Coverage driven
  - Intuitively, if Cov(TS*n*) > max(Cov(TS1), Cov(TS2)),
    add TS*n* to the current population, where *n* = 3 or 4
  - More precisely, the selection is based on a fitness function for branch coverage

Fitness Function for Branch Coverage

- Purpose: Estimates how close a test suite T is to cover all branches
- $f_B(T) = |F| - |F_T| + \sum_{b_k \in B} d(b_k, T) \geq 0$
  - T: the given test suite
  - F: set of all methods
  - F_T: set of methods covered by T
  - B: set of branches
  - $d(b_k, T)$: distance of T from a branch b_k

T is fitter if $f_B(T)$ has a smaller value

Distance Function d(b,T)



if $(\overbrace{x > 10}^{b})$

true

$$d(b,T) = \begin{cases} 0 & \textit{if the branch } b \textit{ has been covered by } T, \\ v(d_{min}(b,T)) & \textit{if the predicate for } b \textit{ has been covered twice by } T, \\ 1 & \textit{otherwise.} \end{cases}$$

- $v(z) = \frac{z}{z+1}$ is a normalizing function with a range (0, 1)
- $d_{min}(b,T)$ is obtained from the minimal value of $d(b,t), t \in T$
- Example of *d(b,t)*:
  - If branch *b* is the true evaluation of predicate ***x* > 10**, and ***x* = 5** in test *t*
  - *d(b,t)* is 10 – 5 + *c* for a small value of *c*, say 0.1

Illustration of $f_B(T) = |F| - |F_T| + \sum_{b_k \in B} d(b_k, T)$

- Suppose T and S are two test suites executing the same set of methods and aim to cover three branches
  - b1: i > 10; b2: j > 10; b3: k > 10
- Suppose T and S each contains three tests
- Tests in T: t1(i=11), t2(j=0), t3(k=0) // one branch covered
  - 3 – 1 + (0 + 0.91 + 0.91) = 3. 82
- Tests in S: s1(i=10), s2(j=10), s3(k=10) // no branch covered
  - 3 – 0 + (0.091 + 0.091 + 0.091) = 3.273
- Evosuite favors S over T in the gene selection

Augmentation with Concolic Testing

- At times, the search algorithm can stuck at finding test suites to reach certain branches $\sum_{b_k \in B} d(b_k, T)$
- Symptom: The components of these branches in the fitness function fail to reduce after a number of iterations
- Solution: Evosuite deploys concolic testing (a.k.a. dynamic symbolic execution) and tries to (partially) solve the concerned predicates using a constraint solver → leading to further reduction in these components

Select the TS with the smallest fitness value

- Evosuite stops generation when
  - It has generated enough Test Suites that altogether satisfy 100% coverage*, OR
  - It has reached the specified time budget
- Select among these test suites the one with the highest coverage (not smallest fitness value)

* It can be configured to branch coverage, exception coverage, weak mutation coverage or strong mutation coverage. Default is branch coverage.

Limitations of Randoop

Coverage saturates quickly with increasing amount of test cases.

Seeds, mutation operators, fitness function

- Generates new test cases randomly. **Coverage-driven**
  - Quality of new test cases is not guaranteed.
- Weak test oracle.
- Not working when handling environment APIs.

Regression &
Mutation-based oracle

API mocking

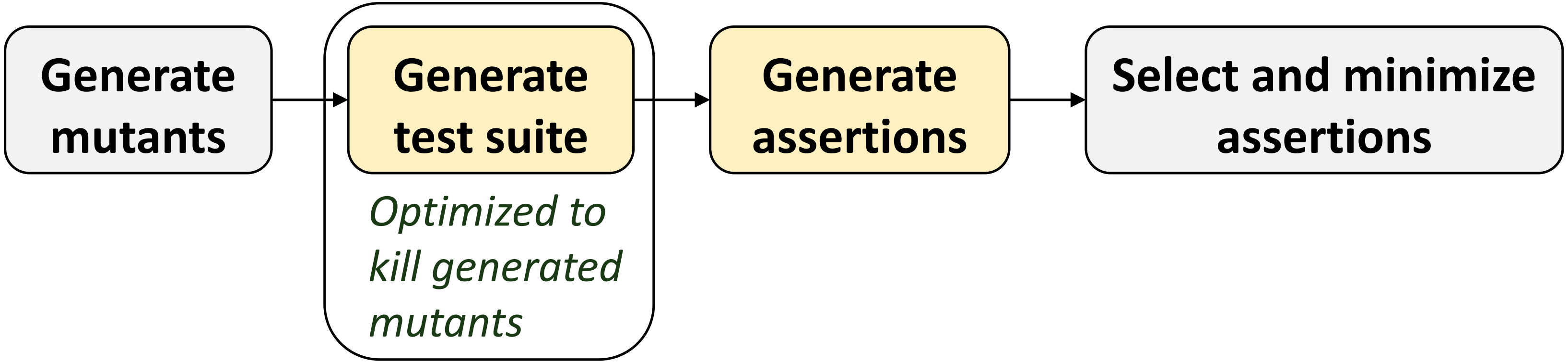
Assertion Generation (Regression-based)

```
public class ClassExampleWithFailure {
  public static int sq(x) {
    return x*x;
  }
  public static int foo(int x, int y) {
    int z = sq(x);
    if (y > 20 && z == 144)
      assert(false);  // assert failure
    return y*z;
  }
}
```

```
@Test(timeout = 4000)
 public void test1()  throws Throwable  {
     int int0 = ClassExampleWithFailure.sq(0);
     assertEquals(0, int0);
 }
…
@Test(timeout = 4000)
 public void test7()  throws Throwable  {
     int int0 = ClassExampleWithFailure.foo(-1158, 0);
     assertEquals(0, int0);
 }
```

Asserts expected regression test outcomes

Assertion Generation (Mutation-based)


Generate mutants
Generate test suite
Optimized to kill generated mutants
Generate assertions
Select and minimize assertions

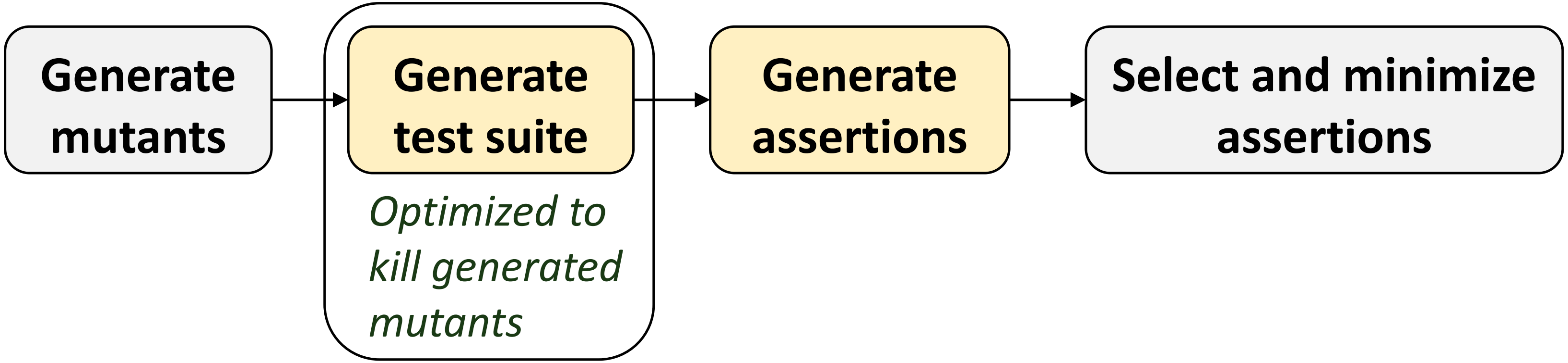
Mutation-driven

- Mutation coverage correlates to fault detection ability
- Can regression tests be generated to achieve high mutation coverage?

statement sequence

```
public void test0() {
  Message v0 = new Message("e");
  Message v1 = new Message("c");
  String v2 = v1.toString();
  assert( … );
}
```

Assertion Generation (Mutation-based)


Generate mutants
Generate test suite
Generate assertions
Select and minimize assertions
Optimized to kill generated mutants

The chicken and egg challenge?

- An assertion can only be generated after deciding the sequence of statements
- But we don't know if a sequence of statements can kill a mutant without deciding the assertion

statement sequence

```
public void test0() {
  Message v0 = new Message("e");
  Message v1 = new Message("c");
  String v2 = v1.toString();
  assert( … );
}
```

Let us revisit this problem in detail later at the Regression Testing topic

Limitations of Randoop

Coverage saturates quickly with increasing amount of test cases.

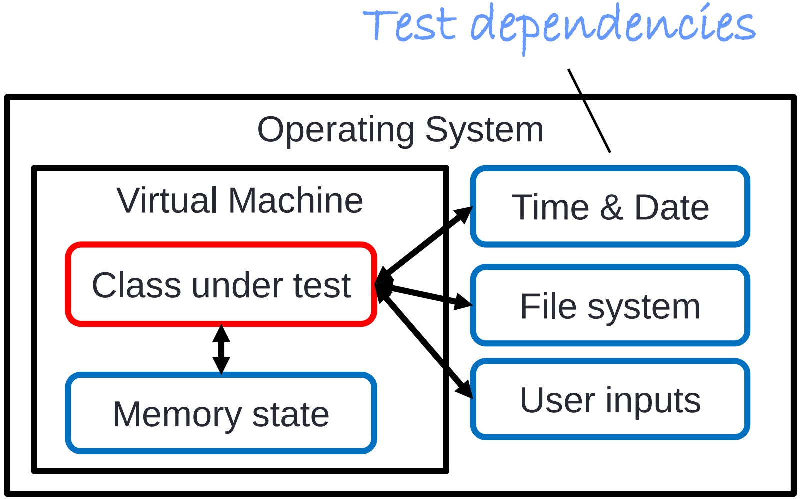
- Generates new test cases randomly. **Coverage-driven**
  - Quality of new test cases is not guaranteed.
- Weak test oracle.
- Not working when handling environment APIs.

Regression & Mutation-based oracle

API mocking

Problems

- The test result of a class interacting with the environment is not consistently reproduced
- Evosuite defines the environment as inputs from outside of a class
  - E.g., a state of virtual machine (free & total memory space), a state of operating system (system time, file system, user inputs).
- A test is *flaky* if some of its executions pass while some fail with the same program and configuration.



Environment API Mocking

- "Mocking" is the replacement of real classes with modified classes that behave consistently.
  - In the following example, `Date` is replaced by `MockDate` that always returns a fixed value.

```
public class Message {
  private Date created;
  private String text;
  public Message(String text) {
    this.created=new MockDate();
    this.text=(text==null?””:contents);
  }
  public String toString() {
    return created+”,”+text;
} }
public class MockDate() {
  public String toString() {
    return “2014.10.02”;
} }
```

```
public class TestSuite1 {
  public void test0() {
    Message v0 = new Message(null);
    String v1 = v0.toString();
    assertFalse(v1.equals(”2014.10.02,null”));
} }
```

Overview of Environment API Mocking

- A test generation tool with a generic mock library creates a non-flaky test suite with higher coverage without user’s (tester) efforts
  - A mock library typically mocks console inputs, file I/O, general API class of Java standard API
    - The mock library consists of a customized `InputStream` class for console inputs, 11 classes of file I/O (e.g. `File`) and 12 general API classes (e.g. `System`, `Runtime`)
    - The mock library has *helper methods* that set the environment in a test case
      - E.g., Mockdate.setdate(Date)
  - Replaces standard library with the mock library using bytecode instrumentation

Console Inputs

- A customized `InputStream` called `SystemInUtil` has a helper method `addInputLine(String)` so that a test case can program the contents of console inputs
  - The console contents of `SystemInUtil` is reset before every test execution.
  - In instrumentation, `System.io` in a class under test is changed into `SystemInUtil.io`

File I/O

- EvoSuite mocks 11 JVM file I/O API classes.
  - Mock classes are subclasses of the 11 API classes to be mocked.
  - EvoSuite overrides the methods of these API classes to access a virtual file system instead of the real file system of operating system.
    - For example, EvoSuite overriddes 37 methods among the 52 methods of `File` class
    - As a result, test cases become independent from each other and there is no negative side-effect such as file system corruption.
  - EvoSuite creates helper methods such as `appendLineToFile()` to control the initial state of the virtual file system.

`java.io.File`	`java.io.PrintStream`
`java.io.FileInputStream`	`java.io.PrintWriter`
`java.io.FileOutputStream`	`java.util.logging.FileHandler`
`java.io.RandomAccessFile`	`javax.swing.JFileChooser`
`java.io.FileReader`	`java.io.FileWriter`
`javax.swing.filechooser.FileSystemView`	

General JVM Calls

- EvoSuite mocks 12 JVM general API classes to control the environment such as time and random number.

Class name	Environment
`java.lang.Exception`	Stack trace message
`java.lang.Throwable`	
`java.util.logging.LogRecord`	
`java.lang.Thread`	
`java.lang.Runtime`	Memory usage & the number of processors
`java.lang.System`	Current system time & date
`java.util.Date`	
`java.util.Calendar`	
`java.util.GregorianCalendar`	
`java.lang.Class`	Reflection (the order of `Method` objects)
`java.lang.Math`	Random number
`java.util.Random`	

Selected Evosuite Parameters

Parameter	Min	Max	Default
Population (test suite) size	5	99	50
Chromosome length	5	99	40
#Mutations	1	10	1
#Initial tests	1	10	10
Crossover rate	0.01	0.99	0.75
Probability of inserting test case	0.01	0.99	0.1
Fitness function	Statement / Branch / Mutation / Exception coverage		Branch coverage

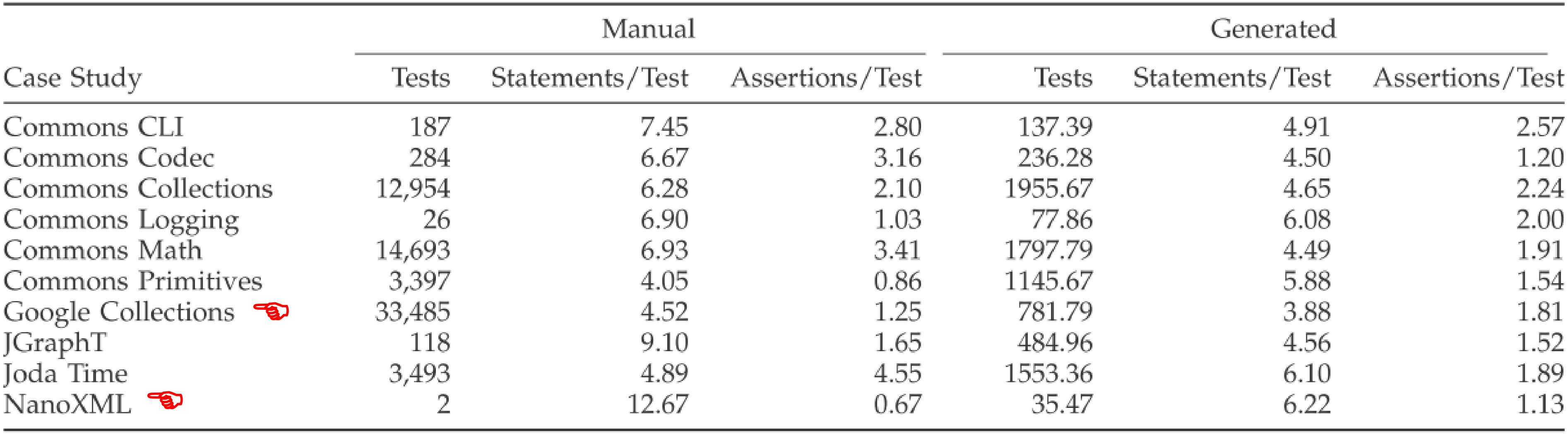
Findings from 100 Java Projects

- Mocking library interacting with the virtual environment increases coverage and reduces flaky tests in automated unit test generation.
  - EvoSuite creates test cases that controls initial environment to increase branch coverage.
  - The generated test cases are non-flaky because interactions with the virtual environment is deterministic.

Are test cases generated by Evosuite useful?
Can Evosuite outperform state-of-the-arts?

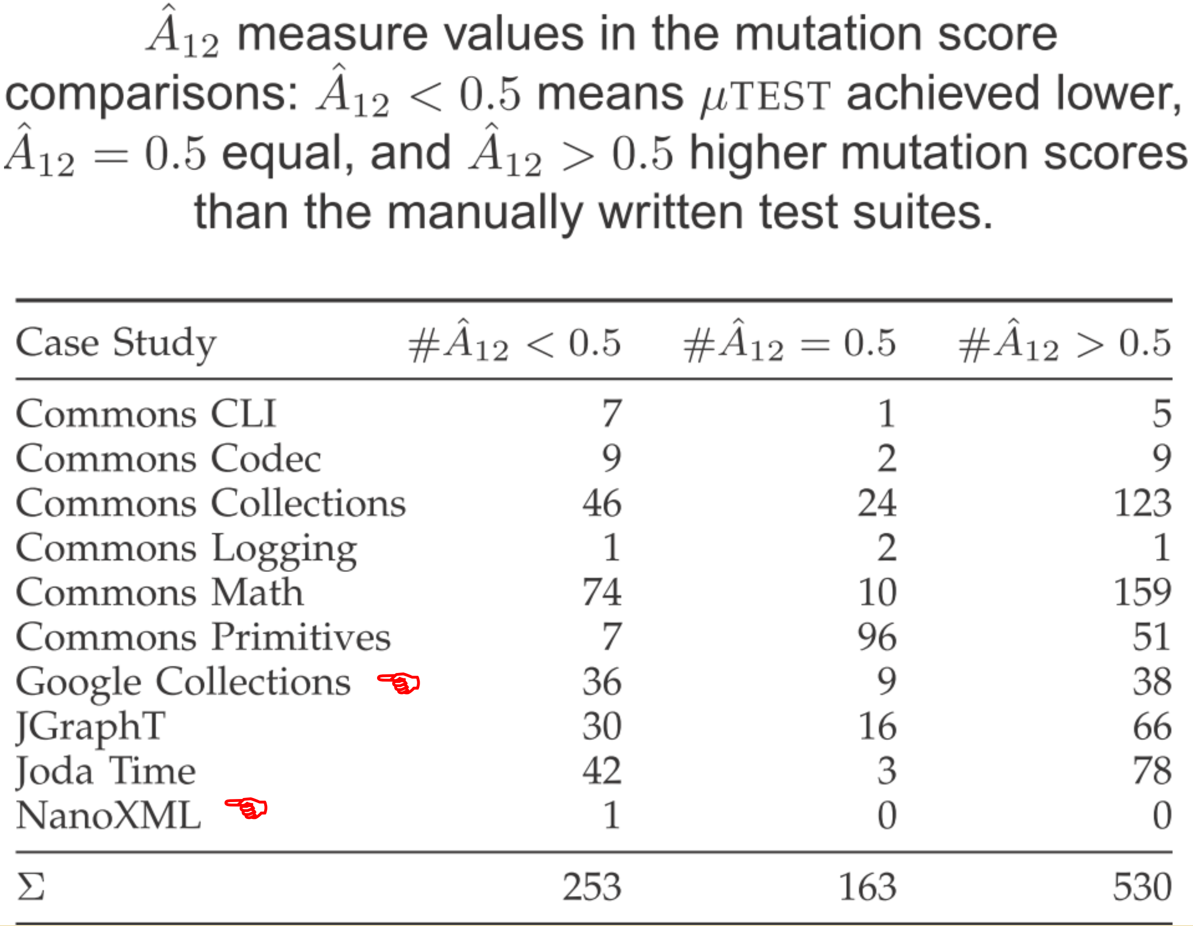
EMPIRICAL STUDY

Total number of test cases and average statistics per test case: Manually handcrafted vs. generated



	Manual			Generated		
Case Study	Tests	Statements/Test	Assertions/Test	Tests	Statements/Test	Assertions/Test
Commons CLI	187	7.45	2.80	137.39	4.91	2.57
Commons Codec	284	6.67	3.16	236.28	4.50	1.20
Commons Collections	12,954	6.28	2.10	1955.67	4.65	2.24
Commons Logging	26	6.90	1.03	77.86	6.08	2.00
Commons Math	14,693	6.93	3.41	1797.79	4.49	1.91
Commons Primitives	3,397	4.05	0.86	1145.67	5.88	1.54
Google Collections ☜	33,485	4.52	1.25	781.79	3.88	1.81
JGraphT	118	9.10	1.65	484.96	4.56	1.52
Joda Time	3,493	4.89	4.55	1553.36	6.10	1.89
NanoXML ☜	2	12.67	0.67	35.47	6.22	1.13

G. Fraser et al., A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite, TOSEM 24 (2), December 2014.



$\hat{A}_{12}$ measure values in the mutation score comparisons: $\hat{A}_{12} < 0.5$ means μTEST achieved lower, $\hat{A}_{12} = 0.5$ equal, and $\hat{A}_{12} > 0.5$ higher mutation scores than the manually written test suites.

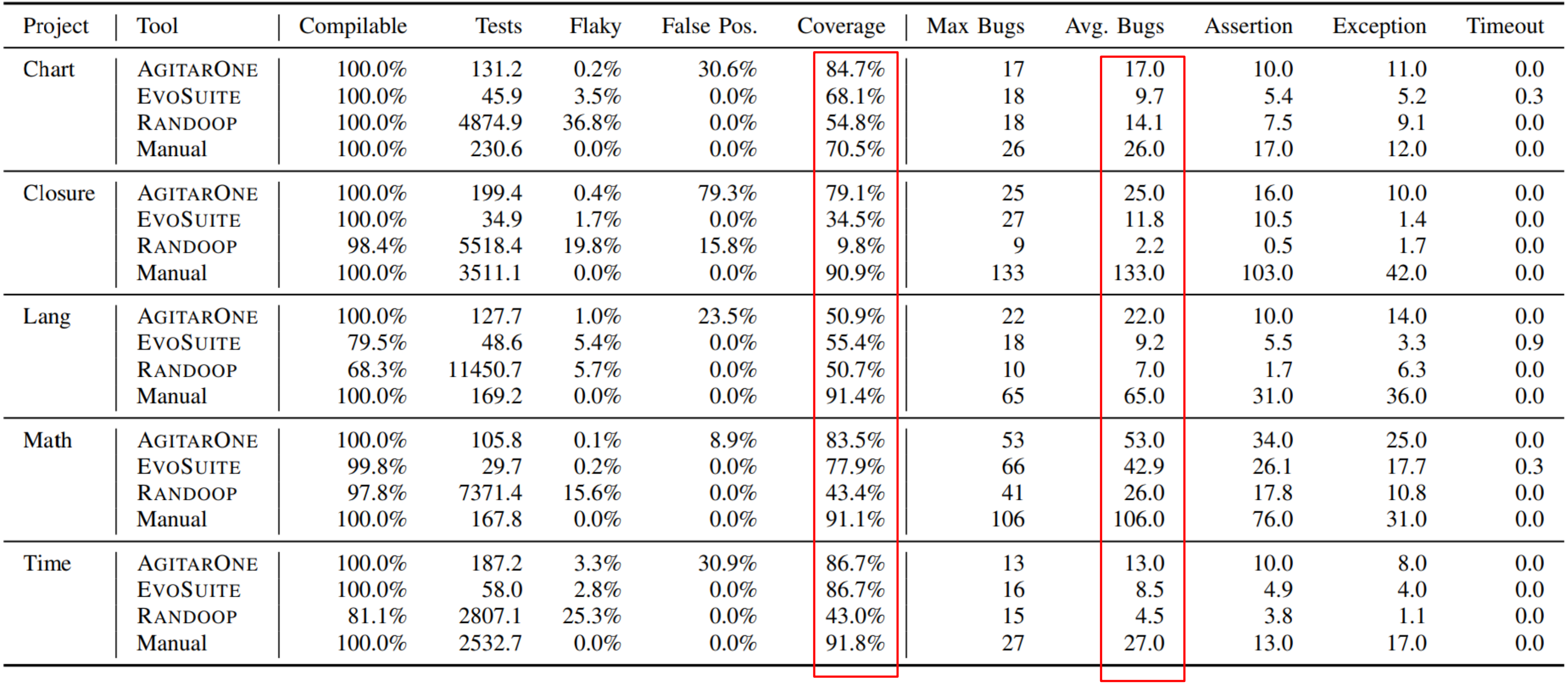
Case Study	$\#\hat{A}_{12} < 0.5$	$\#\hat{A}_{12} = 0.5$	$\#\hat{A}_{12} > 0.5$
Commons CLI	7	1	5
Commons Codec	9	2	9
Commons Collections	46	24	123
Commons Logging	1	2	1
Commons Math	74	10	159
Commons Primitives	7	96	51
Google Collections ☜	36	9	38
JGraphT	30	16	66
Joda Time	42	3	78
NanoXML ☜	1	0	0
Σ	253	163	530

EvoSuite generates test suites and oracles that find significantly more seeded defects than manually written test suites.

Comparison

Source: Sina Shamshiri et al., Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges, ASE15 *// Based on 10 generated test suites*

Regression Bugs



Project	Tool	Compilable	Tests	Flaky	False Pos.	Coverage	Max Bugs	Avg. Bugs	Assertion	Exception	Timeout
Chart	AGITARONE	100.0%	131.2	0.2%	30.6%	84.7%	17	17.0	10.0	11.0	0.0
	EVOSUITE	100.0%	45.9	3.5%	0.0%	68.1%	18	9.7	5.4	5.2	0.3
	RANDOOP	100.0%	4874.9	36.8%	0.0%	54.8%	18	14.1	7.5	9.1	0.0
	Manual	100.0%	230.6	0.0%	0.0%	70.5%	26	26.0	17.0	12.0	0.0
Closure	AGITARONE	100.0%	199.4	0.4%	79.3%	79.1%	25	25.0	16.0	10.0	0.0
	EVOSUITE	100.0%	34.9	1.7%	0.0%	34.5%	27	11.8	10.5	1.4	0.0
	RANDOOP	98.4%	5518.4	19.8%	15.8%	9.8%	9	2.2	0.5	1.7	0.0
	Manual	100.0%	3511.1	0.0%	0.0%	90.9%	133	133.0	103.0	42.0	0.0
Lang	AGITARONE	100.0%	127.7	1.0%	23.5%	50.9%	22	22.0	10.0	14.0	0.0
	EVOSUITE	79.5%	48.6	5.4%	0.0%	55.4%	18	9.2	5.5	3.3	0.9
	RANDOOP	68.3%	11450.7	5.7%	0.0%	50.7%	10	7.0	1.7	6.3	0.0
	Manual	100.0%	169.2	0.0%	0.0%	91.4%	65	65.0	31.0	36.0	0.0
Math	AGITARONE	100.0%	105.8	0.1%	8.9%	83.5%	53	53.0	34.0	25.0	0.0
	EVOSUITE	99.8%	29.7	0.2%	0.0%	77.9%	66	42.9	26.1	17.7	0.3
	RANDOOP	97.8%	7371.4	15.6%	0.0%	43.4%	41	26.0	17.8	10.8	0.0
	Manual	100.0%	167.8	0.0%	0.0%	91.1%	106	106.0	76.0	31.0	0.0
Time	AGITARONE	100.0%	187.2	3.3%	30.9%	86.7%	13	13.0	10.0	8.0	0.0
	EVOSUITE	100.0%	58.0	2.8%	0.0%	86.7%	16	8.5	4.9	4.0	0.0
	RANDOOP	81.1%	2807.1	25.3%	0.0%	43.0%	15	4.5	3.8	1.1	0.0
	Manual	100.0%	2532.7	0.0%	0.0%	91.8%	27	27.0	13.0	17.0	0.0

Interesting Observations

- Randoop generated 21% flaky tests *// largely fixed in latest versions*
- AgitarOne, a commercial product, generated 46% false positives
- Three tools altogether found 55.7% (199 out of 357) bugs
  - No tool alone found more than 40.6% of bugs
- 146 bugs were detected by assertions vs 109 bugs were detected by exceptions; 56 were detected by both
- Only 40% of bugs were detected when their buggy code were covered by generated tests *// not failure-revealing path or variable values*
- Simple bugs were detected by all generated test suites

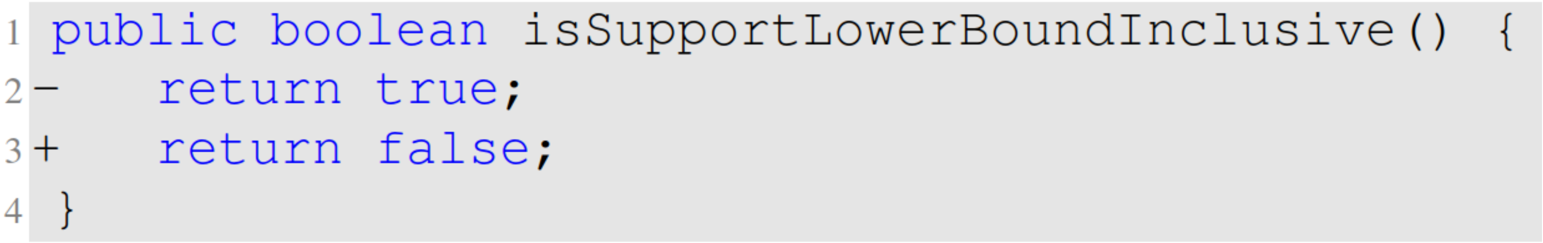
How to improve the effectiveness of generated test cases?

Are there outstanding problems remained to be solved?

OUTSTANDING PROBLEMS

Examples of Simple Bugs

- NullPointerException
- Missing input validation
- Easily executable and observable changes

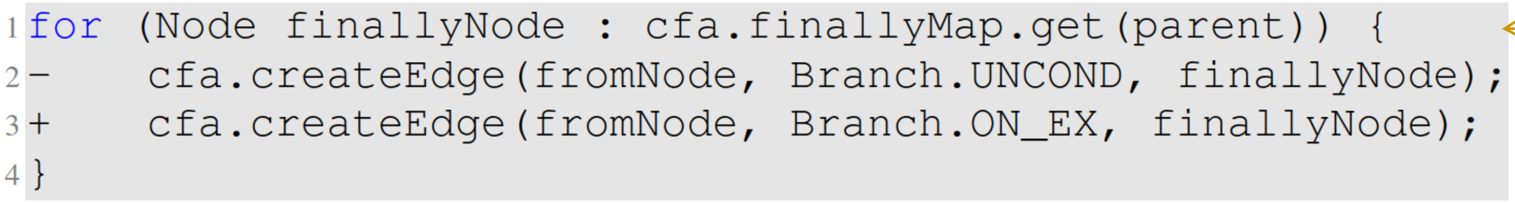


```
 public boolean isSupportLowerBoundInclusive() {
-    return true;
+    return false;
 }
```

Open Problems to Increasing Fault Detection Rate

- Creation of complex objects

Bug fix:

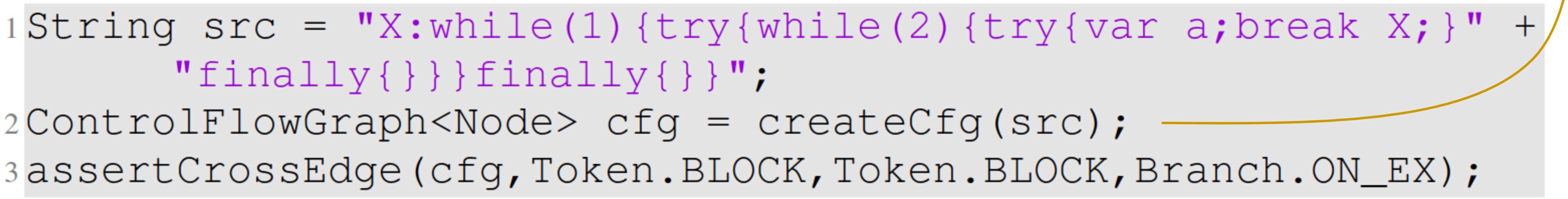


```
for (Node finallyNode : cfa.finallyMap.get(parent)) {
-    cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
+    cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
}
```

calls

- ❑ To detect the bug, a test needs to create a complex string

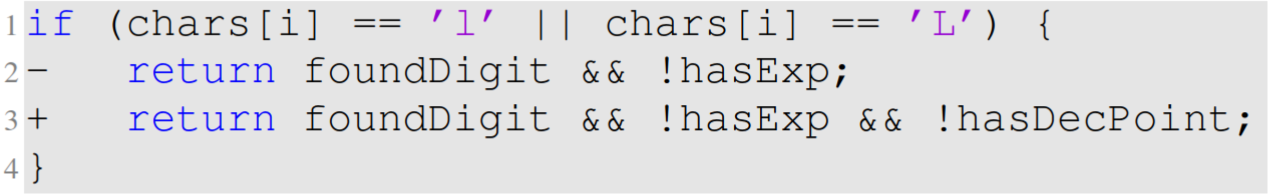
Bug revealing test crafted by developers:



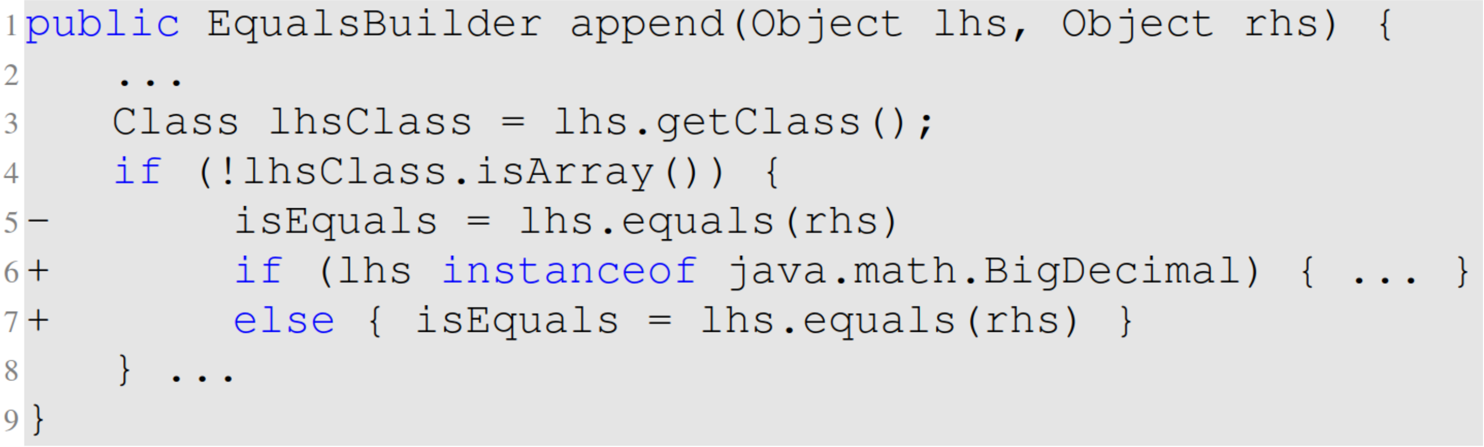
```
String src = "X:while(1){try{while(2){try{var a;break X;}" +
      "finally{}}}finally{}}";
ControlFlowGraph<Node> cfg = createCfg(src);
assertCrossEdge(cfg,Token.BLOCK,Token.BLOCK,Branch.ON_EX);
```

Open Problems to Increasing Fault Detection Rate

- Two examples of complex conditions



```
if (chars[i] == ’l’ || chars[i] == ’L’) {
-   return foundDigit && !hasExp;
+   return foundDigit && !hasExp && !hasDecPoint;
}
```

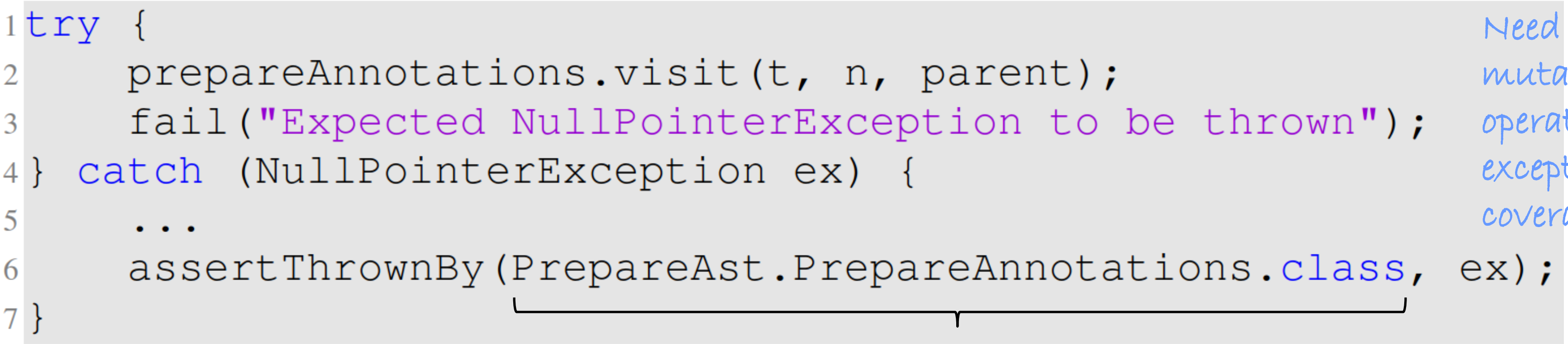


```
public EqualsBuilder append(Object lhs, Object rhs) {
    ...
    Class lhsClass = lhs.getClass();
    if (!lhsClass.isArray()) {
-       isEquals = lhs.equals(rhs)
+       if (lhs instanceof java.math.BigDecimal) { ... }
+       else { isEquals = lhs.equals(rhs) }
    } ...
}
```

Open Problems to Increase Fault Detection Rate

- Generates more complex intra-class data flow dependencies
- Generates stronger assertions
  - Asserts where an expected exception is thrown

fitness on dataflow coverage



```
try {
    prepareAnnotations.visit(t, n, parent);
    fail("Expected NullPointerException to be thrown");
} catch (NullPointerException ex) {
    ...
    assertThrownBy(PrepareAst.PrepareAnnotations.class, ex);
}
```

Specify where the exception is thrown

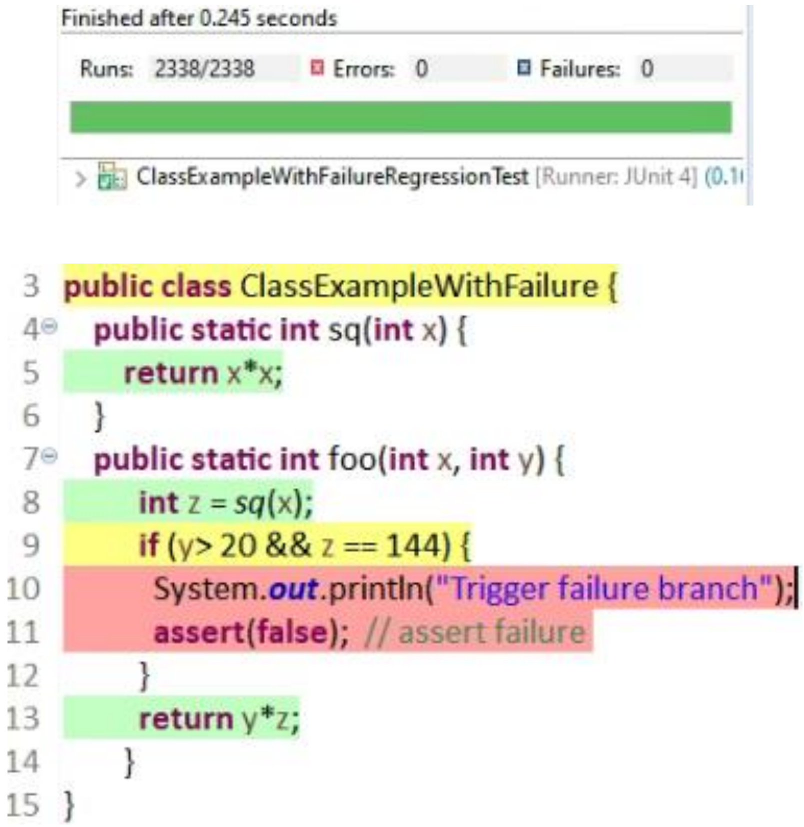
Need better mutation operators for exception coverage

Evosuite

```
public class ClassExampleWithFailure {
        public static int foo(int x, int y) {
                int z = sq(x);
                if (y > 20 && z == 144)
                        assert(false);
                return y*z;
        }
        …
}
```

```
@Test(timeout = 4000)
public void test6()  throws Throwable  {
   try {
     ClassExampleWithFailure.foo(12, 51);
   } catch(AssertionError e) {
      fail("Expecting exception: AssertionError");
   } // …
}

@Test(timeout = 4000)
public void test7()  throws Throwable  {
   int int0 = ClassExampleWithFailure.foo((-1158), 0);
   assertEquals(0, int0);
}
```



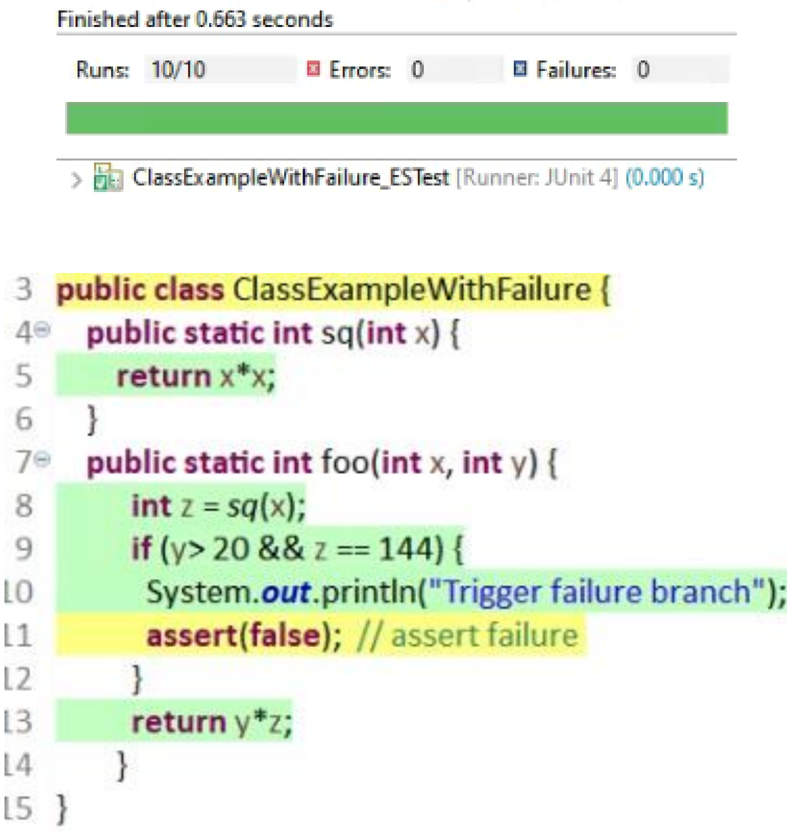
Finished after 0.245 seconds

Runs: 2338/2338 Errors: 0 Failures: 0

ClassExampleWithFailureRegressionTest [Runner: JUnit 4] (0.1

```
public class ClassExampleWithFailure {
  public static int sq(int x) {
    return x*x;
  }
  public static int foo(int x, int y) {
     int z = sq(x);
     if (y> 20 && z == 144) {
      System.out.println("Trigger failure branch");
      assert(false);  // assert failure
     }
     return y*z;
    }
}
```

Coverage by Randoop Generated Tests



Finished after 0.663 seconds

Runs: 10/10 Errors: 0 Failures: 0

ClassExampleWithFailure_ESTest [Runner: JUnit 4] (0.000 s)

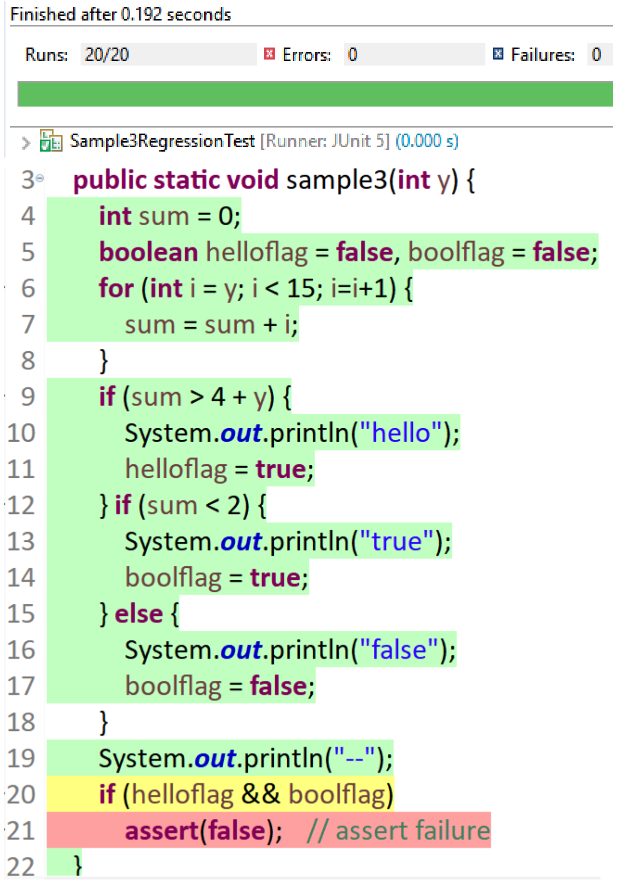
```
public class ClassExampleWithFailure {
  public static int sq(int x) {
    return x*x;
  }
  public static int foo(int x, int y) {
     int z = sq(x);
     if (y> 20 && z == 144) {
      System.out.println("Trigger failure branch");
      assert(false);  // assert failure
     }
     return y*z;
    }
}
```

Coverage by Evosuite Generated Tests

In-Class Exercise 1

Evosuite vs Randoop?

```
public static void sample3(int y) {
        int sum = 0;
        for (int i = y; i < 15; i=i+1) {
            sum = sum + i;
        }
        if (sum > 4 + y)
            System.out.println("hello");
        if (sum < 2) {
            System.out.println("true");
        } else {
            System.out.println("false");
        }
    }
```



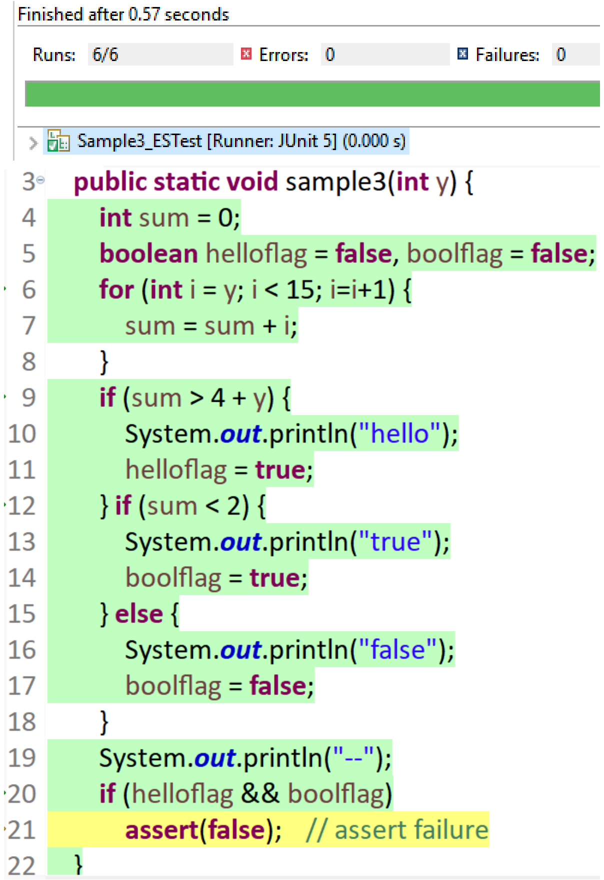
Finished after 0.192 seconds

Runs: 20/20 Errors: 0 Failures: 0

Sample3RegressionTest [Runner: JUnit 5] (0.000 s)

```
public static void sample3(int y) {
    int sum = 0;
    boolean helloflag = false, boolflag = false;
    for (int i = y; i < 15; i=i+1) {
        sum = sum + i;
    }
    if (sum > 4 + y) {
        System.out.println("hello");
        helloflag = true;
    } if (sum < 2) {
        System.out.println("true");
        boolflag = true;
    } else {
        System.out.println("false");
        boolflag = false;
    }
    System.out.println("--");
    if (helloflag && boolflag)
        assert(false);   // assert failure
}
```

Coverage by Randoop Generated Tests



Finished after 0.57 seconds

Runs: 6/6 Errors: 0 Failures: 0

Sample3_ESTest [Runner: JUnit 5] (0.000 s)

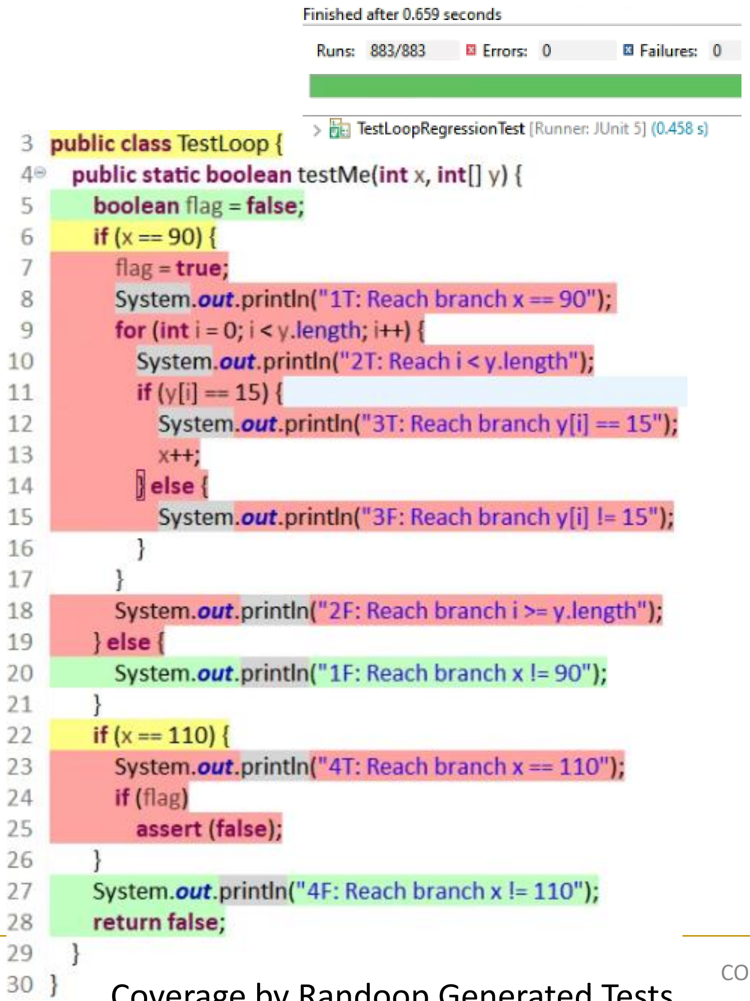
```
public static void sample3(int y) {
    int sum = 0;
    boolean helloflag = false, boolflag = false;
    for (int i = y; i < 15; i=i+1) {
        sum = sum + i;
    }
    if (sum > 4 + y) {
        System.out.println("hello");
        helloflag = true;
    } if (sum < 2) {
        System.out.println("true");
        boolflag = true;
    } else {
        System.out.println("false");
        boolflag = false;
    }
    System.out.println("--");
    if (helloflag && boolflag)
        assert(false);   // assert failure
}
```

Coverage by Evosuite Generated Tests

In-Class Exercise 2 - TestLoop

Evosuite vs Randoop?

```
public static boolean testMe(int x, int[] y) {
        boolean flag = false;
        if (x == 90) {
                flag = true;
                for (int i=0; i<y.length; i++) {
                        if (y[i] == 15) {  x++; } else { }
                }
        } else { }
        if (x == 110) {
                if (flag)
                        assert(false);
        }
        return false;
}
```



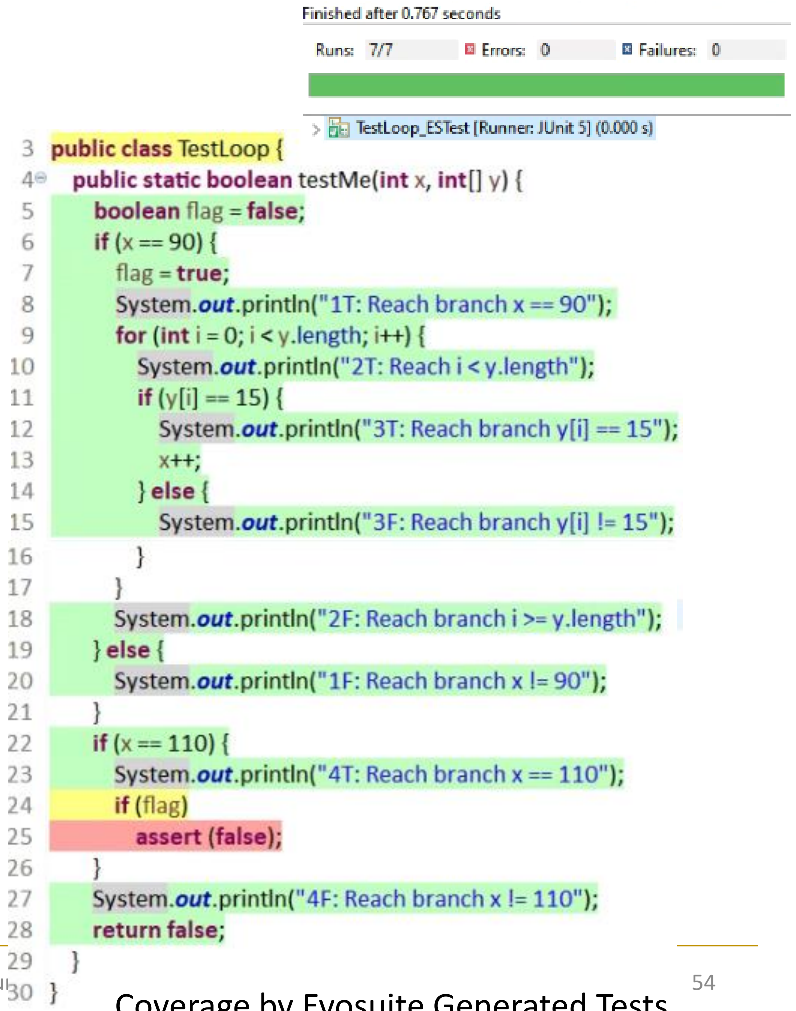
Finished after 0.659 seconds

Runs: 883/883 Errors: 0 Failures: 0

TestLoopRegressionTest [Runner: JUnit 5] (0.458 s)

```
public class TestLoop {
  public static boolean testMe(int x, int[] y) {
    boolean flag = false;
    if (x == 90) {
      flag = true;
      System.out.println("1T: Reach branch x == 90");
      for (int i = 0; i < y.length; i++) {
        System.out.println("2T: Reach i < y.length");
        if (y[i] == 15) {
          System.out.println("3T: Reach branch y[i] == 15");
          x++;
        } else {
          System.out.println("3F: Reach branch y[i] != 15");
        }
      }
      System.out.println("2F: Reach branch i >= y.length");
    } else {
      System.out.println("1F: Reach branch x != 90");
    }
    if (x == 110) {
      System.out.println("4T: Reach branch x == 110");
      if (flag)
        assert (false);
    }
    System.out.println("4F: Reach branch x != 110");
    return false;
  }
}
```

Coverage by Randoop Generated Tests



Finished after 0.767 seconds

Runs: 7/7 Errors: 0 Failures: 0

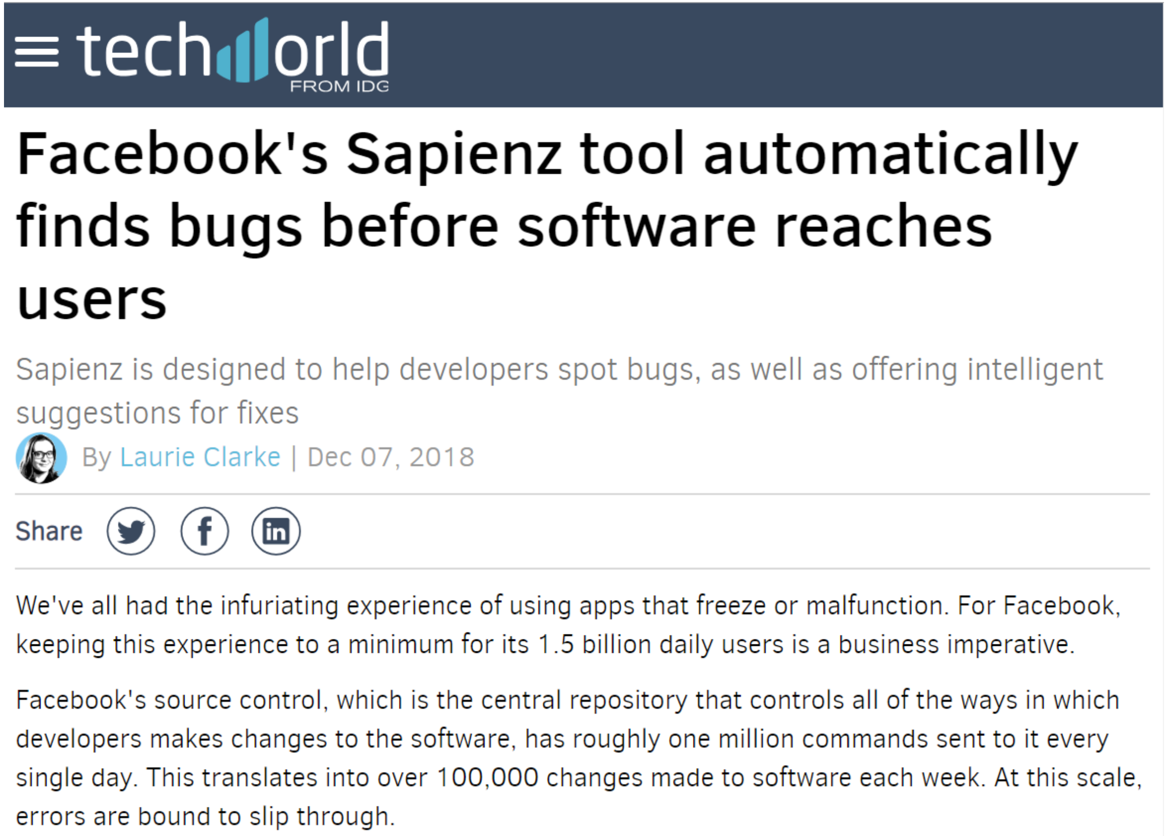
TestLoop_ESTest [Runner: JUnit 5] (0.000 s)

```
public class TestLoop {
  public static boolean testMe(int x, int[] y) {
    boolean flag = false;
    if (x == 90) {
      flag = true;
      System.out.println("1T: Reach branch x == 90");
      for (int i = 0; i < y.length; i++) {
        System.out.println("2T: Reach i < y.length");
        if (y[i] == 15) {
          System.out.println("3T: Reach branch y[i] == 15");
          x++;
        } else {
          System.out.println("3F: Reach branch y[i] != 15");
        }
      }
      System.out.println("2F: Reach branch i >= y.length");
    } else {
      System.out.println("1F: Reach branch x != 90");
    }
    if (x == 110) {
      System.out.println("4T: Reach branch x == 110");
      if (flag)
        assert (false);
    }
    System.out.println("4F: Reach branch x != 110");
    return false;
  }
}
```

Coverage by Evosuite Generated Tests

Adoption of Automated Test Generation at Facebook

Sapienz – Automated Test Generators for Android app



techworld FROM IDG

Facebook's Sapienz tool automatically finds bugs before software reaches users

Sapienz is designed to help developers spot bugs, as well as offering intelligent suggestions for fixes

By Laurie Clarke | Dec 07, 2018

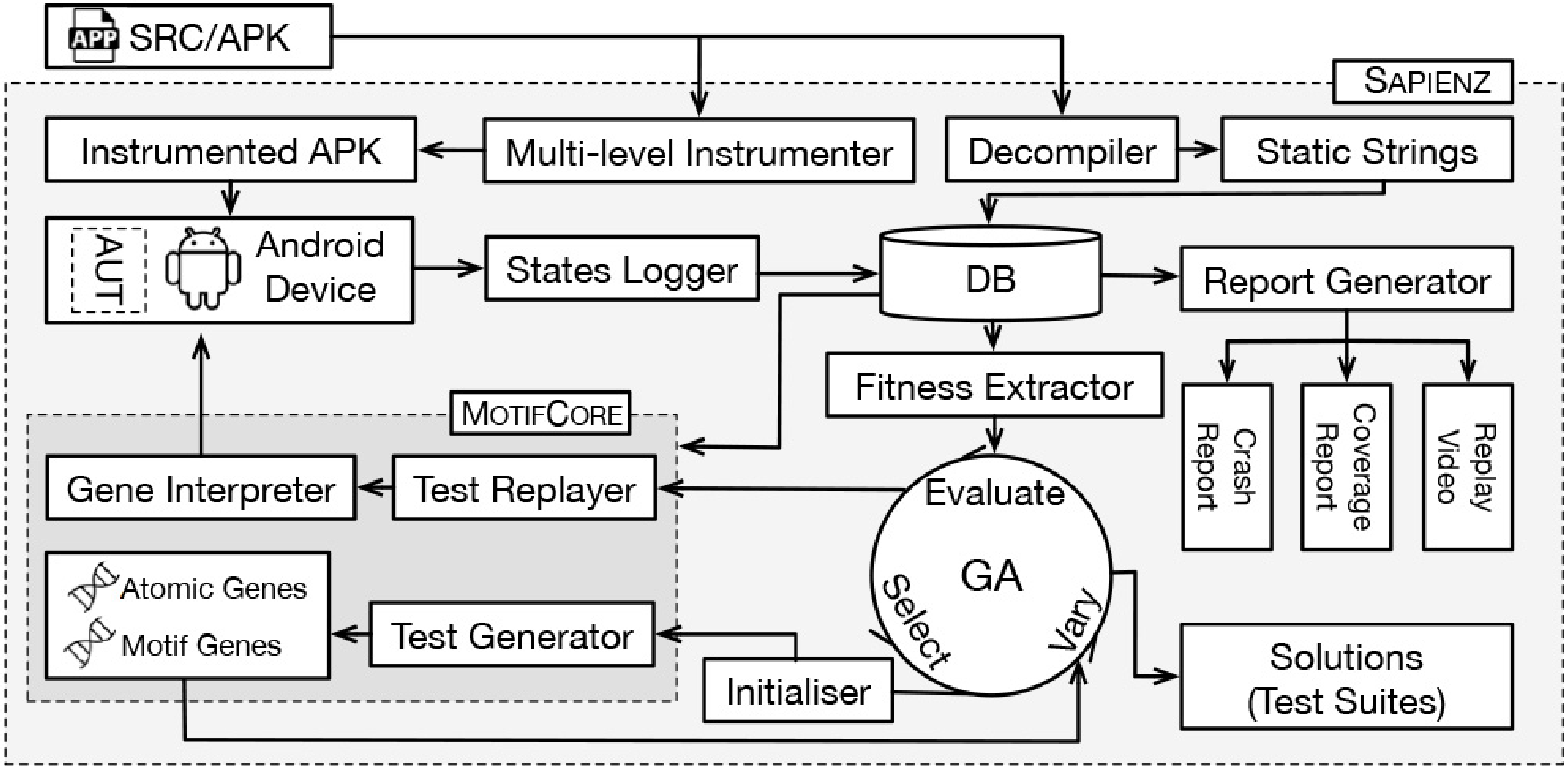
Share

We've all had the infuriating experience of using apps that freeze or malfunction. For Facebook, keeping this experience to a minimum for its 1.5 billion daily users is a business imperative.

Facebook's source control, which is the central repository that controls all of the ways in which developers makes changes to the software, has roughly one million commands sent to it every single day. This translates into over 100,000 changes made to software each week. At this scale, errors are bound to slip through.

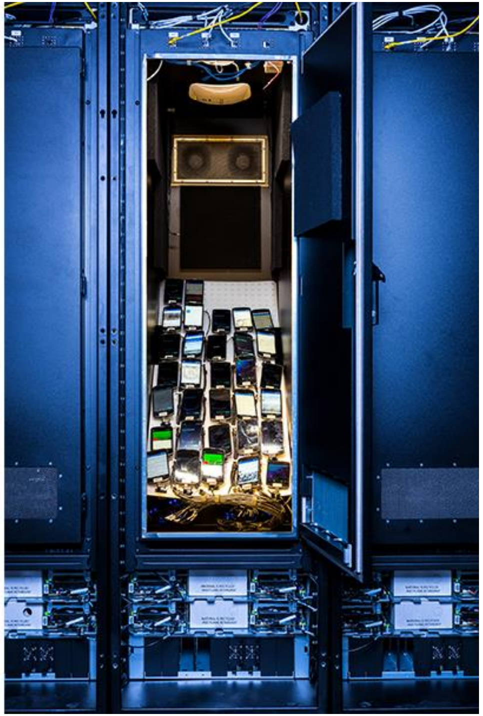
- 100+K code updates weekly at Facebook
- Generates hundreds of monthly bug reports for Facebook, Instagram, Workplace, and Messenger apps
- Pinpointing faulty code

Source: https://www.techworld.com/developers/facebooks-sapienz-tool-automatically-finds-bugs-before-software-reaches-users-3689054/



Sapienz Workflow

Deployment of Sapienz at Facebook



Source: https://arstechnica.com/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz/

Adoption of Sapienz in Facebook

- **Video: Friction Free Fault Finding with Sapienz** 👍
  - https://developers.facebook.com/videos/f8-2018/friction-free-fault-finding-with-sapienz/
- Alshahwan et al., Deploying Search Based Software Engineering with Sapienz at Facebook, SSBSE18 (10th edition)
- Mao et al., Sapienz: Multi-objective Automated Testing for Android Applications, ISSTA16
- Facebook's evolutionary search for crashing software bugs
  - https://arstechnica.com/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz/

Empirical Findings: Automatically Generated (AG) Tests vs Manually Written (MW) Tests

[Almasi et al. ICSE-SEIP 2017]

- coverage(AG tests) > coverage(MW tests): Clearly Yes
  - AG tests can reach codes and branches that are not covered by MW tests
  - AG tests help cover intended behavior not covered by MW tests
- #fault-detected(AG tests) > #fault-detected(MW tests): No evidence
  - Faults detected by AG tests can differ from those detected by MW tests
  - Although AG tests can reach faulty code, the test oracles (i.e., assertions) generated are weak to conclude the test outputs are wrong
- AG tests cannot replace but complement MW tests

Empirical Findings: Automatically Generated (AG) Tests vs Manually Written (MW) Tests

- Commercial software is not necessarily more difficult to cover than open-source software by AG tests
- AG tests can detect as many as 56.4% (Evosuite) and 38.0% (Randoop) of faults in a large-scale financial application
  - Undetected faults requires tests that take either specific input values (50.0%) or complex object configurations (47.6%)
- Outstanding research problems:
  - How to generate stronger test oracles
  - How to generate specific input values and complex object configurations

A recent review by Facebook Engineer on Sapienz in 2020

Testing Apps with Sapienz @ Facebook

Nadia Alshahwan - Sapienz

Software Engineer@facebook

https://youtu.be/BM89PFDwZuU?t=286

Additional References

- G. Fraser and A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, New York, NY, USA, 2011, pp. 416-419.
- F. Gross, G. Fraser, and A. Zeller, “Search-Based System Testing: High Coverage, No False Alarms,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, New York, NY, USA, 2012, pp. 67-77.
- J. Campos, R. Abreu, G. Fraser, and M. d’Amorim, “Entropy-based Test Generation for Improved Fault Localization,” in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2013, pp. 257-267.
- G. Fraser and A. Arcuri, “1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite,” *Empirical Software Engineering*, vol. 20, iss. 3, pp. 611-639, 2013.
- G. Fraser and A. Arcuri, “Whole Test Suite Generation,” *IEEE Transactions on Software Engineering*, vol. 39, iss. 2, pp. 276-291, 2013.
- J. P. Galeotti, G. Fraser, and A. Arcuri, “Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution (Tool paper),” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, New York, NY, USA, 2014, pp. 421-424.
- G. Fraser et al., A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite, TOSEM 24 (2), December 2014.
- G. Fraser et al., Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study, TOSEM 24 (4), August 2015.
- Sina Shamshiri et al., Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges, ASE15
- M.M. Almasi et al., An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application, ICSE-SEIP, 2017.