Joint Degree Program of Fudan University and University College Dublin (UCD)

SOFT620020.02
Advanced Software Engineering

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io

Recap on RANDOOP

- Input
  - A set of classes under test
  - A set of (default or extended) contracts
  - A set of (default or extended) filters
  - Time limit (2 minutes by default)
- Output
  - A set of contract-satisfying test cases
  - A set of contract-violating test cases
- Main idea
  - Generate new test cases by extending previous ones
  - Execute the test case as soon as it is generated
  - Use the execution result to guide test case generation

Course Outline

Date	Topic
Sep. 10	Introduction
Sep. 17	Testing Overview
Sep. 24	Holiday
Oct. 01	Holiday
Oct. 08	Guided Random Testing
Oct. 15	**Search-Based Testing**
Oct. 22	Performance Analysis
Oct. 29	Security Testing

Date	Topic
Nov. 05	Compiler Testing
Nov. 12	Mobile Testing
Nov. 19	Bug Prediction
Nov. 26	Bug Localization
Dec. 03	Delta Debugging
Dec. 10	Automatic Repair
Dec. 17	Symbolic Execution
Dec. 24	Presentation

Whole Test Suite Generation

Gordon Fraser and Andrea Arcuri

TSE 2013, Citation: 273

Discussion 1 – Writing a Test Case to Cover a Specific Branch

Cover the true branch in Line 7

```
Stack stack = new Stack();
stack.push(0);
```

Cover the true branch in Line 11

```
Stack stack = new Stack();
stack.push(0);
stack.pop();
```

Cover the true branch in Line 5

```
Stack stack = new Stack();
stack.push(0); stack.push(0);
stack.push(0); stack.push(0);
```

```
1  public class Stack {
2    int[] values = new int[3];
3    int size = 0;
4    void push(int x) {
5      if(size >= values.length)
6        resize();
7      if(size < values.length)
8        values[size++] = x;
9    }
10   int pop() {
11     if(size > 0)
12       return values[size--];
13     else
14       throw new EmptyStackException();
15   }
16   private void resize(){
17     int[] tmp = new int[values.length * 2];
18     for(int i = 0; i < values.length; i++)
19       tmp[i] = values[i];
20     values = tmp;
21   }
22 }
```

Motivation

```
1 public class Stack {
2   int[] values = new int[3];
3   int size = 0;
4   void push(int x) {
5     if(size >= values.length)
6       resize();
7     if(size < values.length)
8       values[size++] = x;
9   }
10  int pop() {
11    if(size > 0)
12      return values[size--];
13    else
14      throw new EmptyStackException();
15  }
16  private void resize(){
17    int[] tmp = new int[values.length * 2];
18    for(int i = 0; i < values.length; i++)
19      tmp[i] = values[i];
20    values = tmp;
21  }
22 }
```

- Generate one test case for each coverage goal, and then combine them in a single test suite
- Some coverage goals may cover further coverage goals
- Some coverage goals are more difficult to cover than others
- Some coverage goals can be infeasible
- How to dynamically (re-)allocate testing budget to uncovered coverage goals

EVOSUITE

- Whole test suite generation based on genetic algorithms
  - Evolve all the test cases in a test suite at the same time
  - Consider all coverage goals at the same time
  - Goal: cover all coverage goals at the same time while keeping the size as small as possible when automated testing oracles are not available

- Large-scale evaluations on 1,741 public classes
  - 19 open-source libraries and one industrial case study
  - Comparisons with a single branch approach

Discussion 2 – Writing Minimal Test Cases to Cover All Feasible Branches

```
Stack stack = new Stack();
try{
    stack.pop();
} catch(EmptyStackException e) {}
stack.push(0); stack.push(0);
stack.push(0); stack.push(0);
stack.pop();
```

```
1  public class Stack {
2    int[] values = new int[3];
3    int size = 0;
4    void push(int x) {
5      if(size >= values.length)
6        resize();
7      if(size < values.length)
8        values[size++] = x;
9    }
10   int pop() {
11     if(size > 0)
12       return values[size--];
13     else
14       throw new EmptyStackException();
15   }
16   private void resize(){
17     int[] tmp = new int[values.length * 2];
18     for(int i = 0; i < values.length; i++)
19       tmp[i] = values[i];
20     values = tmp;
21   }
22 }
```

Preliminaries on Genetic Algorithms

- Genetic algorithm is an optimization technique to find an optimal or sub-optimal solution to an optimization problem

- Inspired by the biological evolution process
- Use concepts of “Natural Selection” and “Genetic Inheritance” (Darwin 1859)
- Originally developed by John Holland (1975)

- Basic idea: maintain a population of candidate solutions for the problem at hand, and evolve it by iteratively applying a set of genetic operators

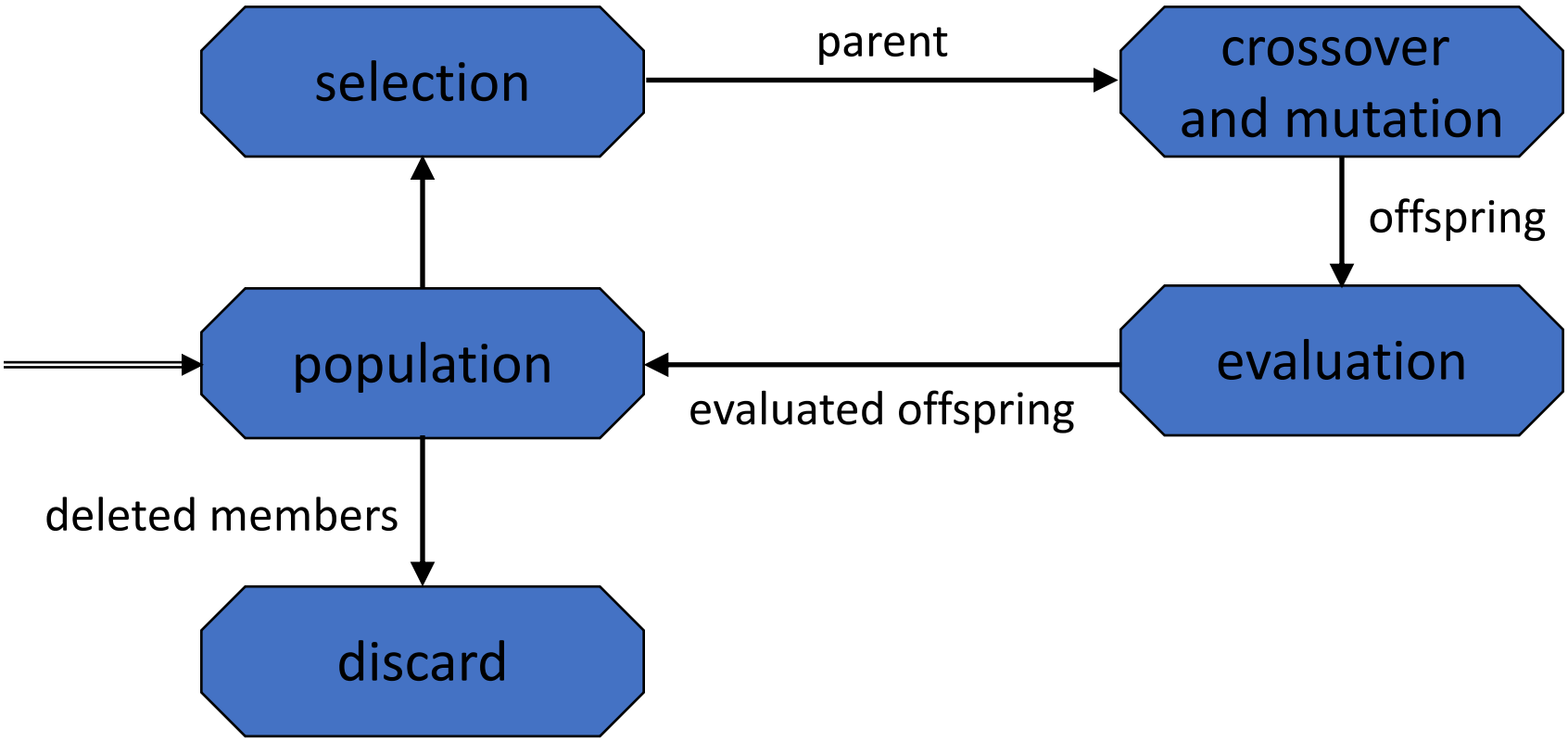
The Metaphor

Genetic Algorithm	Nature
Optimization Problem	Environment
Feasible Solutions	Individuals living in the environment
Solution quality (or fitness function)	Individual's degree of adaptation to the surrounding environment
A set of feasible solutions	A population of species
Genetic operators	Selection, crossover, and mutation in nature's evolutionary process
Iteratively applying a set of genetic operators on a set of feasible solutions	Evolution of populations to suit their environment

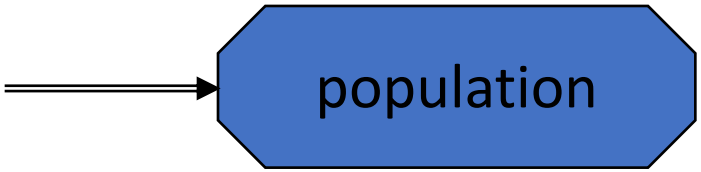
Simple Genetic Algorithm

produce an initial population of individuals;
evaluate the fitness of all individuals;
while termination condition not met **do**
    select fitter parent individuals;
    crossover and mutate parent individuals;
    evaluate the fitness of the offspring individuals;
    generate a new population;
end while

The Evolutionary Cycle in GA



Initial Population and Evaluation



- Representation of an individual
  - bit strings, e.g., 0 1 1 0 … 1 1 0 0
  - integer numbers, e.g., 43 -33 24 -9 ... 0 10 89 -12
  - ...
- Generation of the initial population of size *n*
  - e.g., random generation
- Fitness function of evaluating an individual
  - problem-specific

An Example: the MAXONE Problem

- The problem: find a string that maximizes the number of ones in a string of *m* binary digits

- Suppose m = 10 and n = 6
- Toss a fair coin 60 times to generate the initial population

s1 = 1111010101 f(s1) = 7

s2 = 0111000101 f(s2) = 5

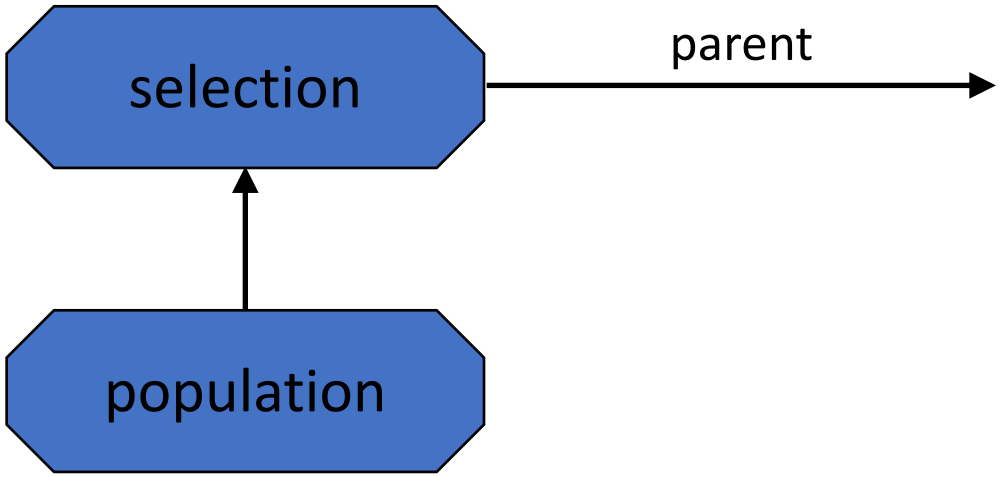
s3 = 1110110101 f(s3) = 7

s4 = 0100010011 f(s4) = 4

s5 = 1110111101 f(s5) = 8

s6 = 0100110000 f(s6) = 3

Selection



- Two parent individuals are selected with probabilities biased in relation to their fitness values
  - e.g., tournament selection (select k individuals from the population at random and select the best to become a parent)

An Example of Selection

- The initial population

  s1 = 1111010101 f(s1) = 7

  s2 = 0111000101 f(s2) = 5

  s3 = 1110110101 f(s3) = 7

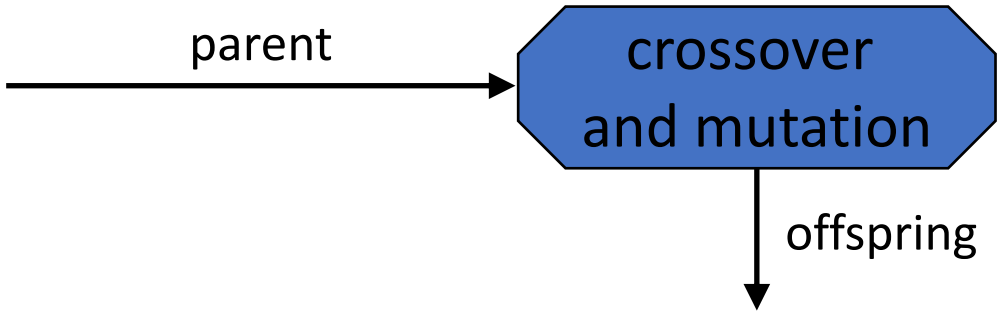
  s4 = 0100010011 f(s4) = 4

  s5 = 1110111101 f(s5) = 8

  s6 = 0100110000 f(s6) = 3

- Parent 1: s1, s2, s6 ⇨ s1
- Parent 2: s3, s4, s5 ⇨ s5

Crossover and Mutation



- Crossover: combine the genetic information of two parent to generate new offspring
  - e.g., single point crossover (randomly pick a point, and swap the right of that point between the two parent)
- Mutation: change one or more gene values in an individual to ensure the diversity from one population to the next
  - e.g., bit flip mutation (randomly pick a point and flip the bit)

An Example of Crossover and Mutation

- Two selected parent: s1 = 1111010101, s5 = 1110111101

- Single point crossover

  s1 = 1111010101 ⇨ s1’ = 1111111101

  s5 = 1110111101 ⇨ s5’ = 1110010101

- Bit flip mutation

  s1’ = 1111111101 ⇨ s1’’ = 1111111111

  s5’ = 1110010101 ⇨ s5’’ = 1110000101

Discussion 3 – Selection, Crossover and Mutation

s2 = 0111000101, f(s2) = 5
s4 = 0100010011, f(s4) = 4
s6 = 0100110000, f(s6) = 3

s2 = 0111000101, f(s2) = 5
s3 = 1110110101, f(s3) = 7
s4 = 0100010011, f(s4) = 4

Tournament selection

s2 = 0111000101, f(s2) = 5
s3 = 1110110101, f(s3) = 7

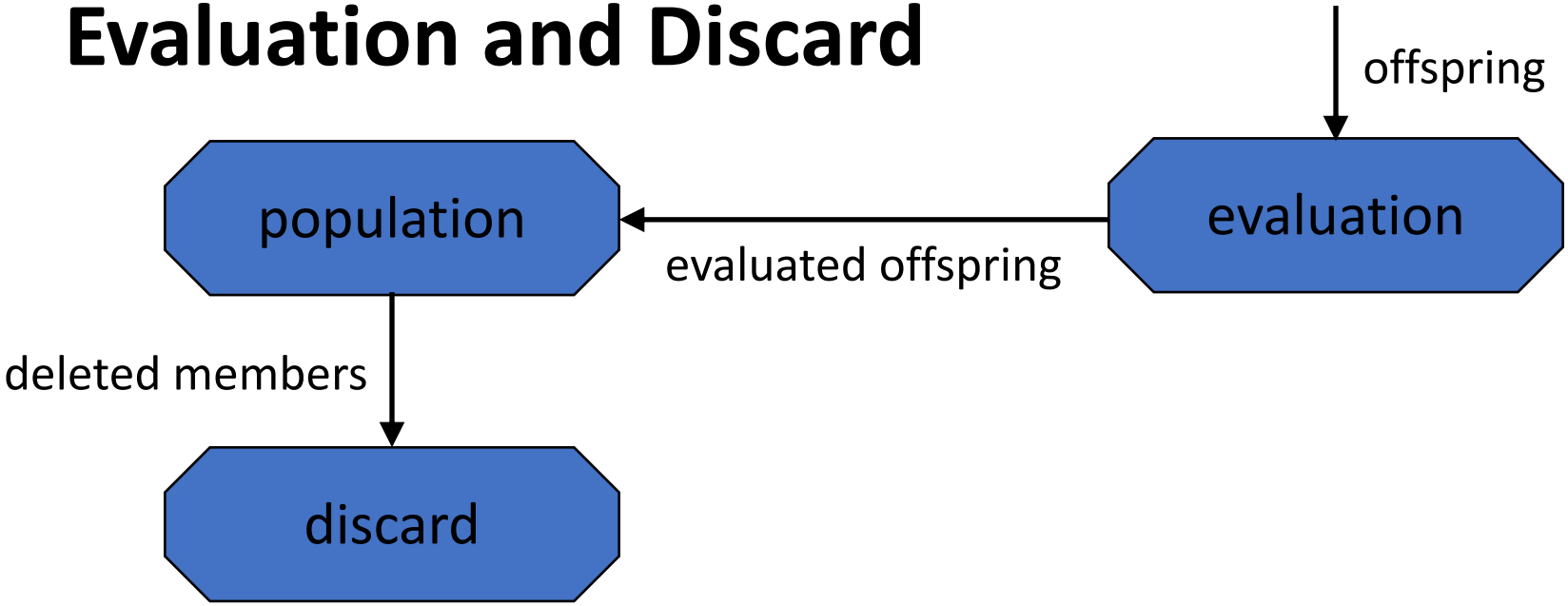
Single point crossover at point 4

s2 = 0111000101 ⇨ s2’ = 0110110101
s3 = 1110110101 ⇨ s3’ = 1111000101

Bit flip mutation at point 9

s2’ = 0110110101 ⇨ s2’’ = 0110110111
s3’ = 1111000101 ⇨ s3’’ = 1111000111

Evaluation and Discard



- Discard: select the individuals for the next generation
  - Generational GA: replace entire population
  - Steady-state GA: replace a few members with good ones

GA in EVOSUITE

Algorithm 1. The genetic algorithm applied in EVOSUITE

```
1  current_population ← generate random population
2  repeat
3      Z ← elite of current_population
4      while |Z| ≠ |current_population| do
5          P1, P2 ← select two parents with rank selection
6          if crossover probability then
7              O1, O2 ← crossover P1, P2
8          else
9              O1, O2 ← P1, P2
10         mutate O1 and O2
11         fP = min(fitness(P1), fitness(P2))
12         fO = min(fitness(O1), fitness(O2))
13         lP = length(P1) + length(P2)
14         lO = length(O1) + length(O2)
15         TB = best individual of current_population
16         if fO < fP ∨ (fO = fP ∧ lO ≤ lP) then
17             for O in {O1, O2} do
18                 if length(O) ≤ 2 × length(TB) then
19                     Z ← Z ∪ {O}
20                 else
21                     Z ← Z ∪ {P1 or P2}
22         else
23             Z ← Z ∪ {P1, P2}
24     current_population ← Z
25 until solution found or maximum resources spent
```

Generate the initial population

Add the best to the new population

Select two parent

Crossover on two parent

Mutate to generate two offspring

Generate the new population

Apply selection, crossover, mutation and evaluation to fill the new population

Add good offspring or parent to the new population

Representation of an Individual

- An individual (or a solution) in the whole test suite generation is a test suite $T = \{t_1, t_2, \ldots, t_n\}$ whose size $|T|$ is n
- A test case is a sequence of statements $\langle s_1, s_2, \ldots, s_l \rangle$ of length l
- Each statement s_i in a test case represents one value $v(s_i)$

```
Stack stack = new Stack();      s1
int val = 1;                    s2
stack.push(val);                s3

Stack stack = new Stack();      s1
int val = 0;                    s2
stack.push(val);                s3
stack.pop();                    s4
```

Fitness Function

- Branch coverage is considered in EVOSUITE
  - B: the set of branches in a system under test
  - M: the set of methods in a system under test

- An optimal solution is defined as a solution that
  - cover all the feasible branches/methods (first goal)
  - minimal in the total number of statements (second goal)

- If two test suites have the same coverage, the selection mechanism rewards the test suite with fewer statements, i.e., the shorter one (Line 3, 5 and 18 in the algorithm)

Fitness Function (cont.)

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k, T)$$

$$d(b, T) = \begin{cases} 0, & \text{if the branch has been covered,} \\ \nu(d_{min}(b, T)), & \text{if the predicate has been executed at least twice,} \\ 1, & \text{otherwise.} \end{cases}$$

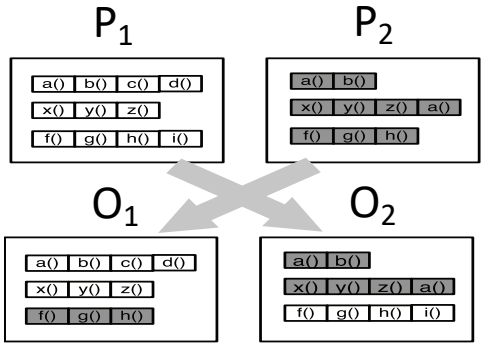
```
Stack stack = new Stack();
int val = 1;
stack.push(val);
stack.push(val);

Stack stack = new Stack();
int val = 0;
stack.push(val);
stack.pop();
```

```
1  public class Stack {
2    int[] values = new int[3];
3    int size = 0;
4    void push(int x) {
5      if(size >= values.length)
6        resize();
7      if(size < values.length)
8        values[size++] = x;
9    }
10   int pop() {
11     if(size > 0)
12       return values[size--];
13     else
14       throw new EmptyStackException();
15   }
16   private void resize(){
17     int[] tmp = new int[values.length * 2];
18     for(int i = 0; i < values.length; i++)
19       tmp[i] = values[i];
20     values = tmp;
21   }
22 }
```

Genetic Operators – Crossover

- Two parent individuals: P_1 and P_2
- $O_1 = \alpha|P_1| + (1 - \alpha)|P_2|$
- $O_2 = \alpha|P_2| + (1 - \alpha)|P_1|$




```
Stack stack = new Stack();
int v = 1;
stack.push(v); stack.push(v);
stack.push(v); stack.push(v);
stack.pop();

Stack stack = new Stack();
int v = 0;
stack.push(v);
stack.pop();
```

P_1

```
Stack stack = new Stack();
int v = 1;
stack.push(v);
stack.push(v);
stack.pop();

Stack stack = new Stack();
try {
   stack.pop();
} catch () {};
```

P_2

```
Stack stack = new Stack();
int v = 1;
stack.push(v); stack.push(v);
stack.push(v); stack.push(v);
stack.pop();

Stack stack = new Stack();
try {
   stack.pop();
} catch () {};
```

O_1

Genetic Operators – Mutation

- Test suite mutation: mutate each test case in T with probability $1/|T|$, i.e., on average, only one test case is mutated; and then add new random test cases with probability σ, σ^2, …
- Test case mutation: apply remove, change and insert in order on $t = \langle s_1, s_2, \ldots, s_l \rangle$ (each is applied with probability 1/3)

Test Case Mutation – Remove

- Delete each statement s_i with probability $1/l$ (on average, only one statement is deleted)
- s_j ($i+1 <= j <= l$) might use $v(s_i)$
  - replace the use of $v(s_i)$ with a value from $\{v(s_k) \mid 0 <= k < j \text{ and } k != i\}$ which has the same type of $v(s_i)$
  - or delete s_j as well recursively

```
Stack stack = new Stack();
int v = 1;
stack.push(v);
stack.push(v);
stack.push(v);
stack.push(v);
stack.pop();
```

```
Stack stack = new Stack();
int v = 1;
stack.push(v);
stack.push(v);
stack.push(v);
stack.push(v);
stack.pop();
```

```
Stack stack = new Stack();
int v1 = 1;
int v2 = 0;
stack.push(v2);
stack.push(v2);
stack.push(v2);
stack.push(v2);
stack.pop();
```

Discussion 4 – Mutation with Remove

```
Stack s1 = new Stack();
Stack s2 = new Stack();
int v = 1;
s1.push(v);
s1.push(v);
s2.push(v);
s2.push(v);
s2.pop();
```

```
Stack stack = new Stack();
int v1 = 1;
boolean b = stack.push(v1);
int v2 = b ? -1 : 1;
stack.push(v2);
stack.pop();
```

Test Case Mutation – Change

- Change each statement s_i with probability 1/l (on average, only one statement is changed)
  - s_i is primitive statement: change the numeric value by a random value, or change the string by deleting, changing or inserting a char
  - s_i is an assignment statement: either the variable on the left-hand or right-hand side of the assignment is replaced with a different variable of the same type
  - otherwise, replace s_i with a method, field, or constructor with the same return type as $v(s_i)$ and the parameters satisfiable with the values in the set $\{v(s_k) \mid 0 <= k < i\}$

```
Stack stack = new Stack();
int v1 = 3;
int v2 = 0;
stack.push(v1);
stack.push(v2);
stack.pop();
```

```
Stack stack = new Stack();
int v1 = 1;
int v2 = 1;
stack.push(v1);
stack.push(v2);
stack.pop();
```

```
Stack stack = new Stack();
int v1 = 1;
int v2 = 0;
stack.push(v1);
stack.push(v1);
stack.pop();
```

Discussion 5 – Mutation with Change

```
Stack s1 = new Stack();
Stack s2 = new Stack();
int v = 1;
s1.push(v);
s1.push(v);
s2.push(v);
s2.push(v);
s2.pop();
```

```
Stack s1 = new Stack();
Stack s2 = new Stack();
int v = 1;
s1.push(v);
s1.push(v);
s2.push(v);
s2.push(v);
s2.pop();
```

```
Stack stack = new Stack();
int v1 = 1;
int v2 = 2;
int v3;
v3 = v2;
stack.push(v1);
stack.push(v2);
stack.push(v3);
stack.pop();
```

Test Case Mutation – Insert

- Insert new statements at a random position i with probability β, β^2, … For each statement insertion, with probability 1/3
  - insert a random call of the class under test or its member classes
  - insert a method call on a value in the set $\{v(s_k) \mid 0 <= k < i\}$
  - use a value in $\{v(s_k) \mid 0 <= k < i\}$ as a parameter in a call of the class under test or its member classes

```
Stack s1 = new Stack();
int v = 1;
s1.push(v);
Stack s2 = new Stack();
s1.push(v);
s1.push(v);
```

```
Stack s1 = new Stack();
int v = 1;
s1.push(v);
s1.pop();
s1.push(v);
s1.push(v);
```

```
Stack s1 = new Stack();
int v = 1;
s1.push(v);
s1.push(v);
s1.push(v);
s1.push(v);
```

Generating Random Test Cases

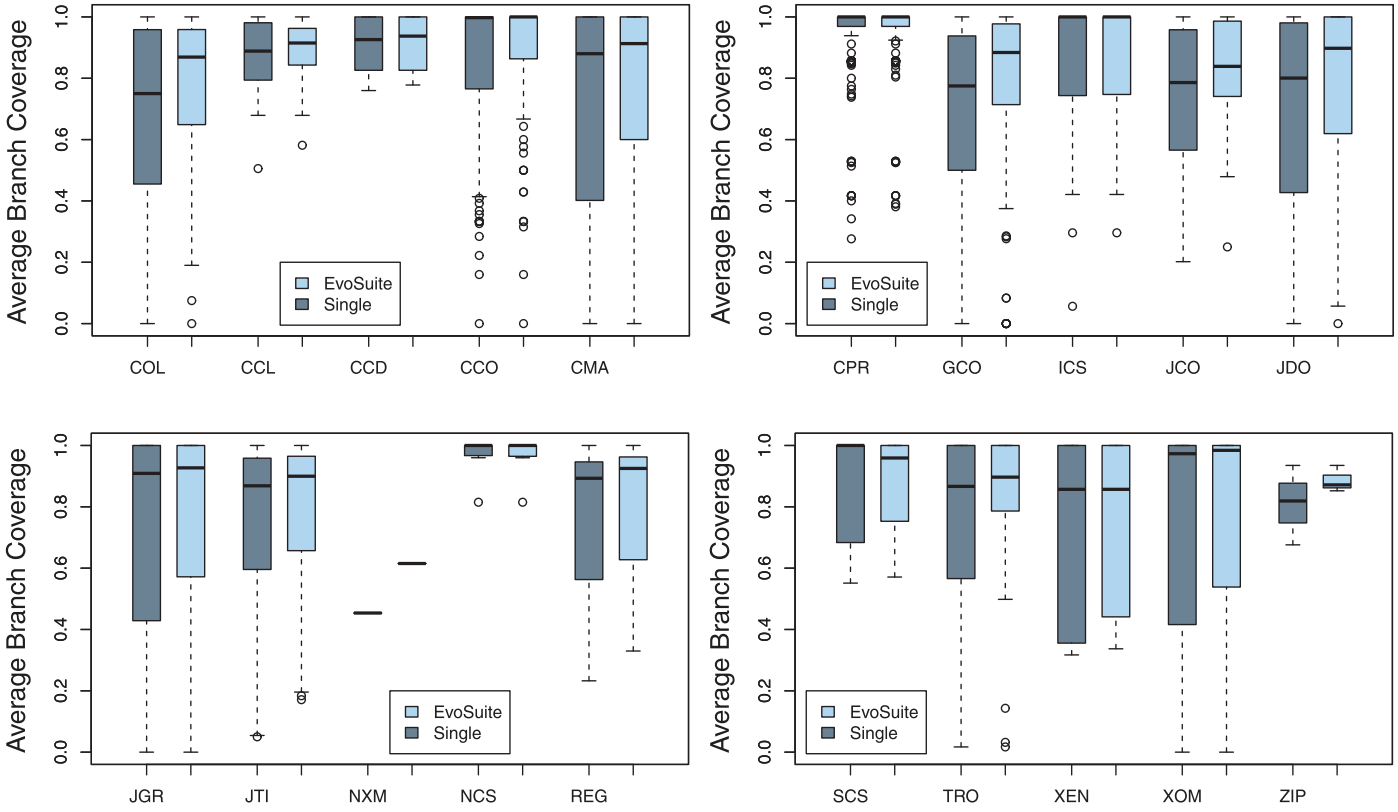
- Choose a random value r between 1 and L, and repeatedly apply the insertion operator used in mutation until r statements are generated

Evaluation – Subjects

Case Study		#Classes		#Branches	LOC[1]
		Public	All		
COL	Colt	135	298	10,795	20,741
CCL	Commons CLI	14	15	662	1,078
CCD	Commons Codec	21	22	1,369	2,205
CCO	Commons Collections	246	421	8,683	19,190
CMA	Commons Math	247	306	10,503	23,881
CPR	Commons Primitives	210	231	2,874	7,008
GCO	Google Collections	85	370	4,214	9,886
ICS	Industrial Casestudy	21	29	373	809
JCO	Java Collections	30	118	3,531	6,339
JDO	JDom	57	61	4,098	6,452
JGR	JGraphT	137	193	2,467	5,924
JTI	Joda Time	131	199	8,681	18,003
NXM	NanoXML	1	1	310	661
NCS	Numerical Casestudy	11	11	209	421
REG	Java Regular Expressions	3	91	1,922	3,020
SCS	String Casestudy	12	12	607	606
TRO	GNU Trove	205	591	10,585	24,297
XEN	Xmlenc	7	7	1,645	788
XOM	XML Object Model	165	185	11,794	23,814
ZIP	Java ZIP Utils	3	4	219	441
Σ		1,741	3,165	85,541	175,564

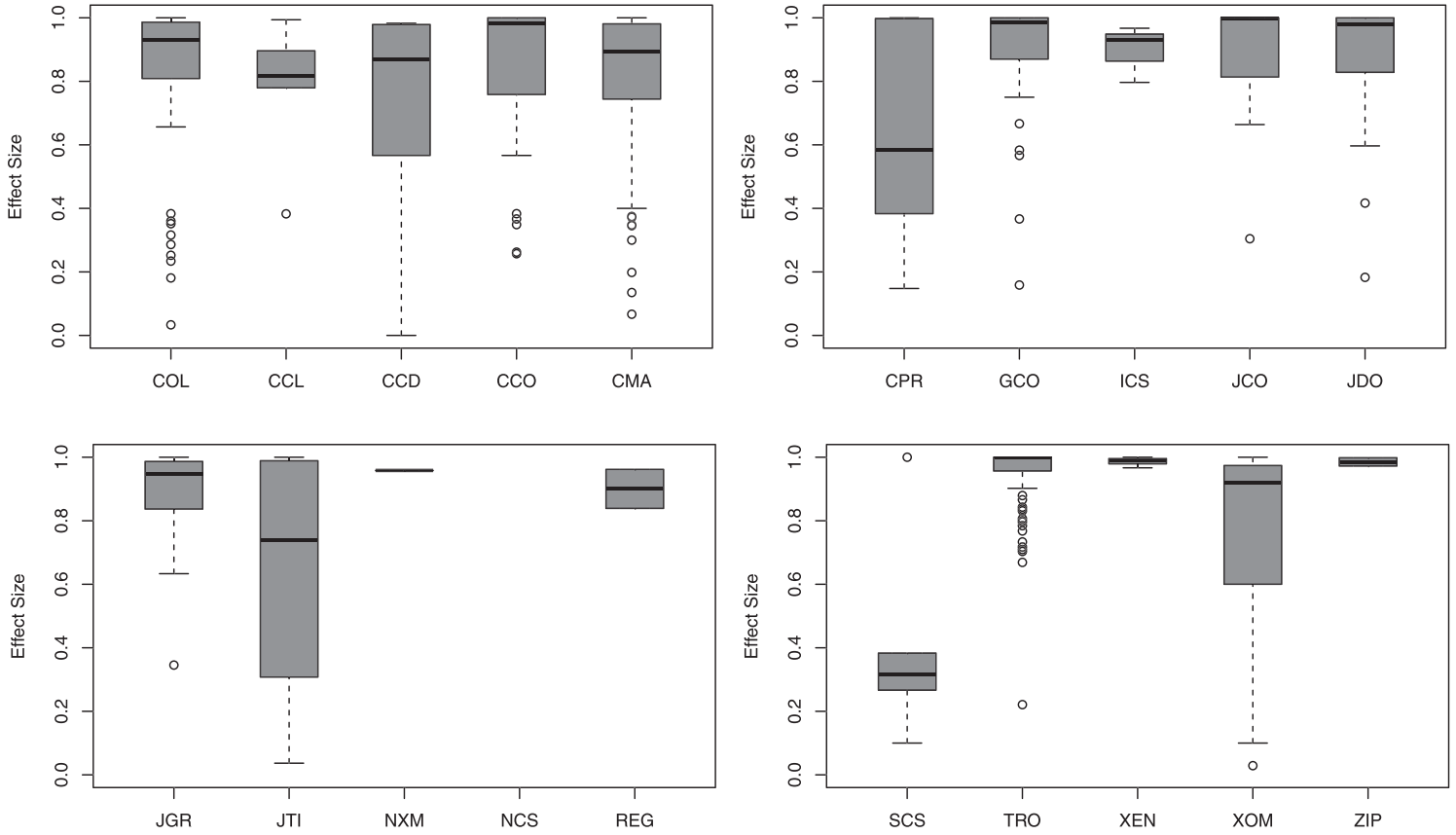
Repeat the comparison between EVOSUITE and the single branch approach for 30 times

Evaluation – Branch Coverage



On average, EVOSUITE obtained 83% coverage, whereas the single branch approach obtained 77%

Evaluation – Branch Coverage (cont.)



A high value of effect size means that, in all of the 30 comparisons, EVOSUITE obtained higher coverage than the single branch approach

Evaluation – Branch Coverage (cont.)

$\hat{A}_{12}$ Measure Values in the Coverage Comparisons: $\hat{A}_{12} < 0.5$ Means EvoSuite Resulted in Less, $\hat{A}_{12} = 0.5$ Equal, and $\hat{A}_{12} > 0.5$ Better Coverage Than a Single Branch Approach

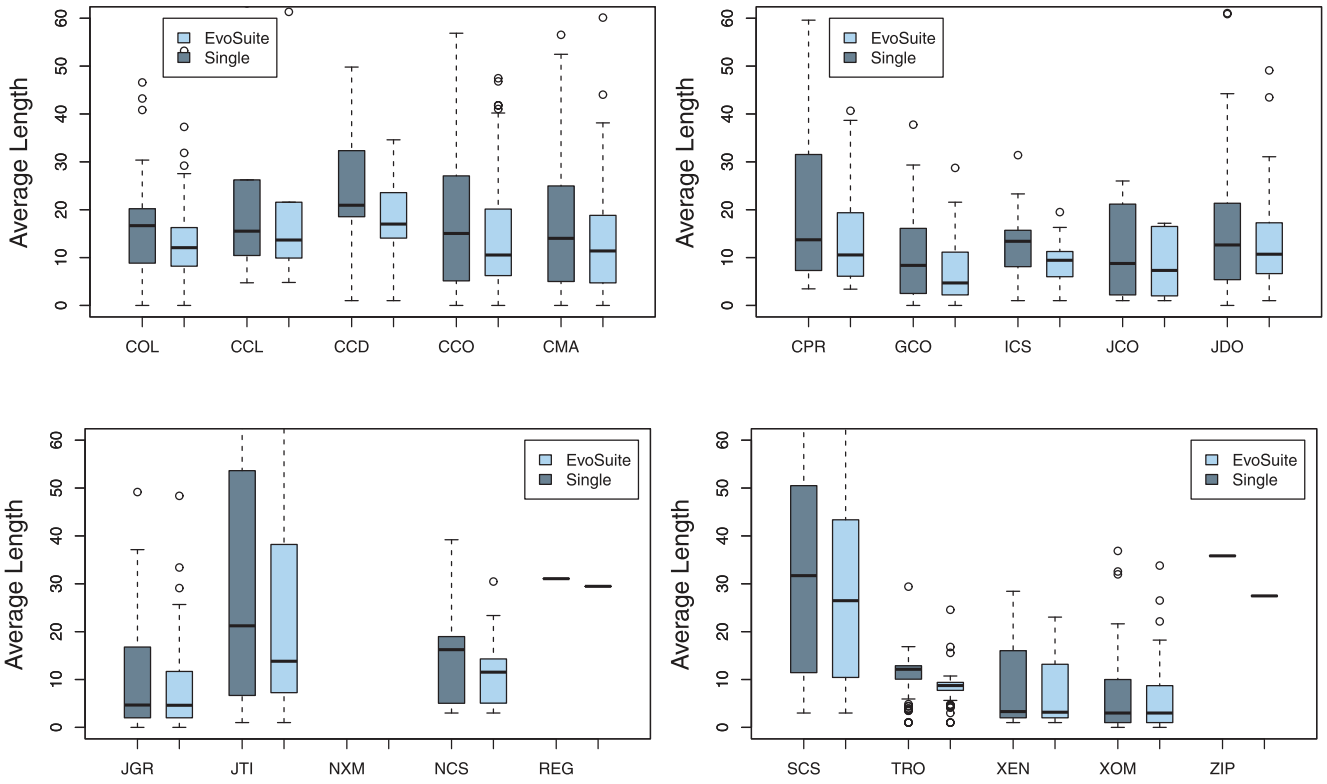
Case Study	$\#\hat{A}_{12} < 0.5$	$\#\hat{A}_{12} = 0.5$	$\#\hat{A}_{12} > 0.5$
COL	13(9)	30	92(79)
CCL	2(1)	6	6(4)
CCD	2(1)	13	6(5)
CCO	19(5)	137	90(81)
CMA	24(10)	100	123(103)
CPR	23(10)	150	37(19)
GCO	4(2)	31	50(42)
ICS	0(0)	17	4(3)
JCO	2(1)	10	18(17)
JDO	3(2)	27	27(25)
JGR	2(1)	88	47(41)
JTI	41(28)	28	62(41)
NXM	0(0)	0	1(1)
NCS	1(0)	10	0(0)
REG	0(0)	1	2(2)
SCS	4(4)	6	2(1)
TRO	2(1)	73	130(124)
XEN	0(0)	4	3(3)
XOM	22(11)	92	51(45)
ZIP	0(0)	1	2(2)
Σ	164(86)	824	753(638)

In parentheses "()" the number of times the effect size is statistically significant at level 0.05.

When EVOSUITE is worse on a specific SUT, it is only worse by a little, whereas when it is better, it is better by a larger quantity

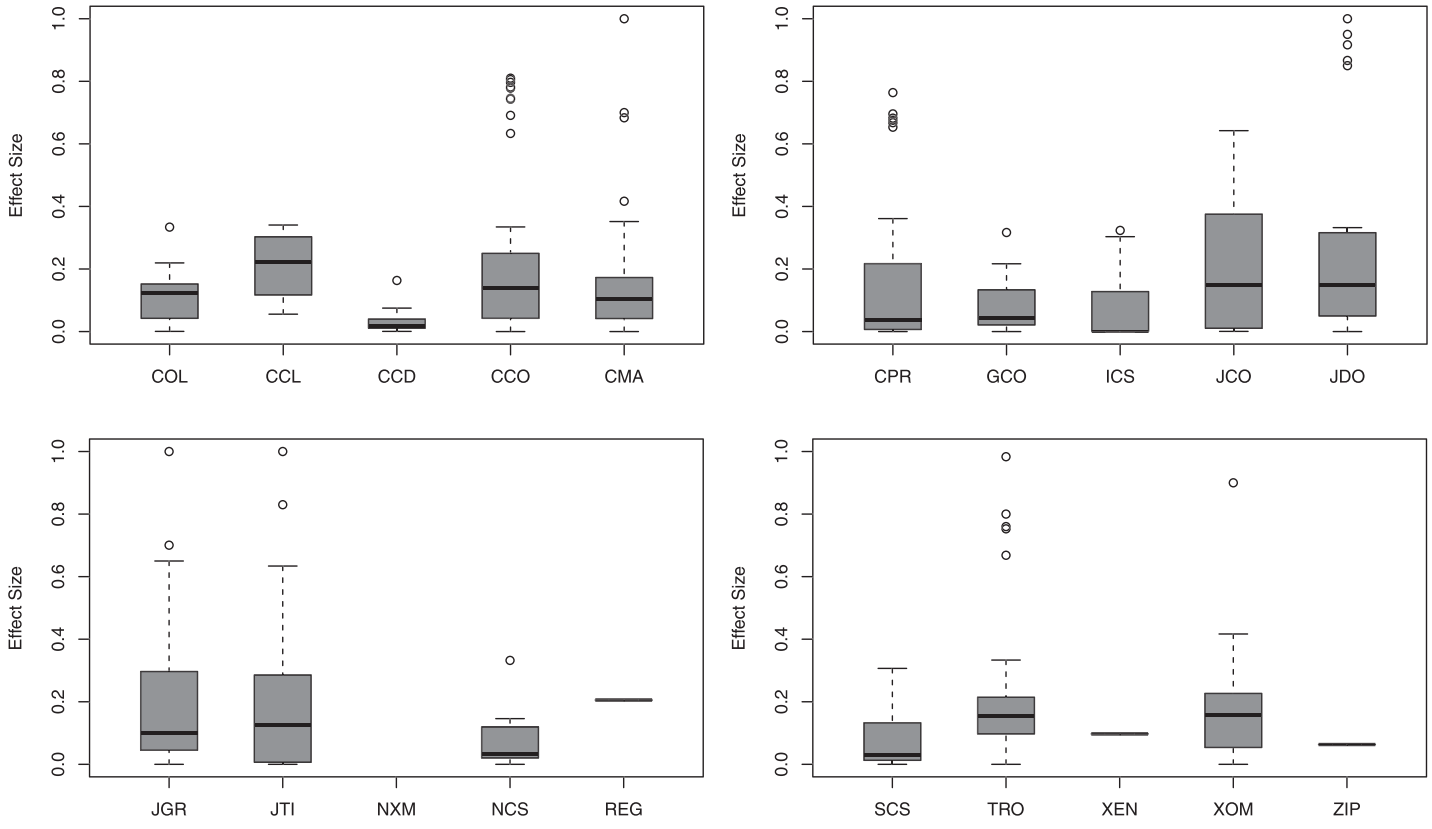
In 824 classes, EVOSUITE and the single branch approach achieved the same coverage. In such cases, it is important to analyze what is the resulting size of the generated test suites

Evaluation – Test Suite Size



EVOSUITE had an average length of 3.23 statements, whereas the the single branch approach had an average length of 8.36

Evaluation – Test Suite Size (cont.)



A low value of effect size means that, in all of the 30 comparisons, EVOSUITE had a low probability of generating longer test suites than the single branch approach

Discussion 6 – 100% Coverage?

Why does EVOSUITE not achieve 100% coverage for all classes?

Infeasible branches

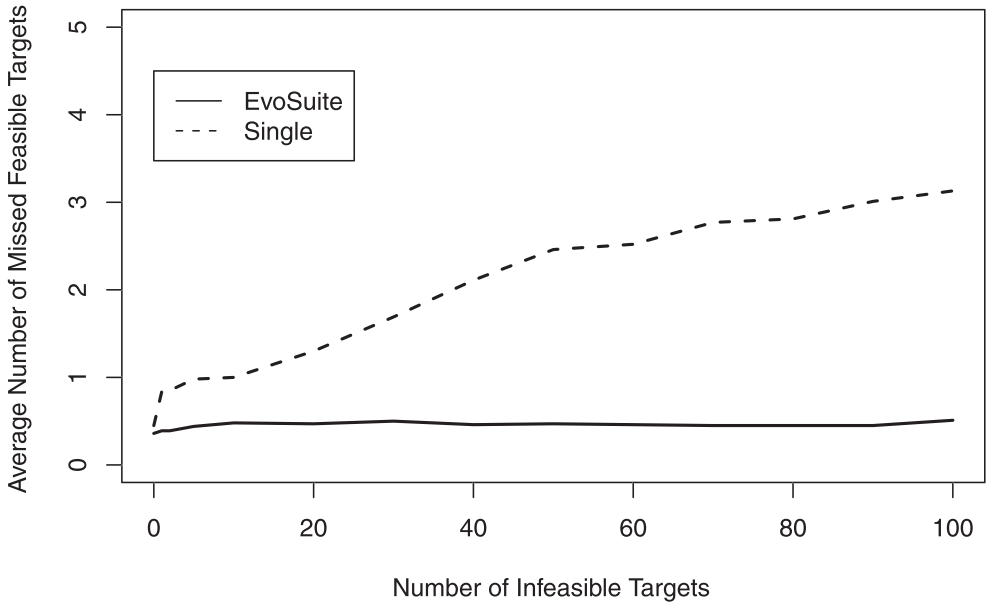
- private methods are not called in any public methods
- dead code
- methods of abstract classes are overridden in all concrete subclasses without calling the abstract superclass
- …

Difficult branches

- Environment dependency (e.g., network connection, file system)
- Multithreaded code (hard to handle thread creation and termination)
- Anonymous and private classes (only handled indirectly via the owner classes' public interface)
- …

Evaluation – Infeasible Coverage Goals

```
1 class Infeasible {
2   void infeasibleGoals(int x, int y) {
3     if(x > 0 && y > 0 && 2 * x == Math.sqrt(y)) {
4     }
5     // Infeasible branches are added here
6   }
7 }
```



Conclusions

- Approach and evaluation to demonstrate that optimizing whole test suites toward all coverage goals at the same time is superior to the traditional approach of targeting one coverage goal at a time

- To learn more about EVOSUITE, visit the website at http://www.evosuite.org

Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges

Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn and Andrea Arcuri

ASE 2015, ACM SIGSOFT Distinguished Paper Award

Methodology

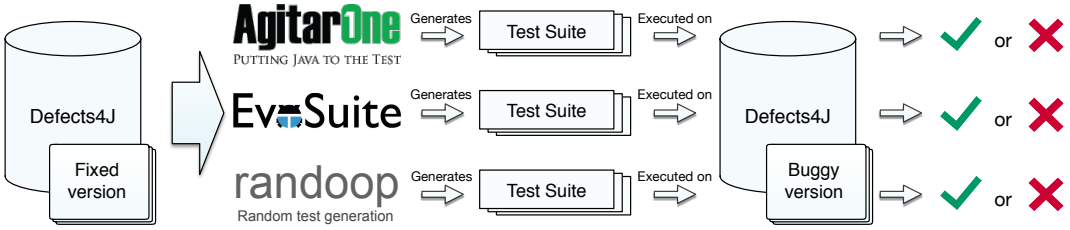


Fig. 1: Overview of the experimental setup. For each fault, the Defects4J dataset provides a *buggy* (i.e., faulty) and a *fixed* version. Test suites are generated with all tools on the fixed version, and executed on the buggy version.

Defects4J consists of 357 real faults from five open source projects: *JFreeChart* (26), *Google Closure compiler* (133), *Apache Commons Lang* (65), *Apache Commons Math* (106), and *Joda Time* (27)

RQ1: how do automated unit test generators perform at finding faults?

RQ2: how do automated unit test generators need to be improved to find more faults?

Results for RQ1

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

Automated test generation tools found 55.7% of the bugs, but no tool alone found more than 40.6%.

Results for RQ2

- Manual analysis of undetected faults
  - 12 faults that no tool managed to cover (not even partially)
  - 4 undetected faults that were fully covered
  - 76 faults that only one tool managed to detect (28 by AGITARONE, 35 by EVOSUITE and 13 by RANDOOP)

Results for RQ2 (cont.)

- Improving coverage
  - Complex objects
  - Complex conditions
  - Private methods/fields
- Improving propagation and detection
  - Propagation
  - Assertion generation
- Flaky tests
  - Environments dependencies
  - Static state

Reading Materials

- Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. IEEE Trans. Softw. Eng. 39, 2 (2013), 276–291.
- S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. 2015 Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In ASE. 201–211.
- Gordon Fraser and Andrea Arcuri. 2012. The seed is strong: Seeding strategies in search-based software testing. In ICST. 121–130.
- José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding Strategies in Search-based Unit Test Generation. Softw. Test. Verif. Reliab. 26, 5 (2016), 366–401.
- Andrea Arcuri, Gordon Fraser, and J. P. Galeotti. 2014. Automated Unit Test Generation for Classes with Environment Dependencies. In ASE. 79–90.
- Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2010. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. IEEE Trans. Softw. Eng. 36, 6 (2010), 742–762.
- Mark Harman and Phil McMinn. 2010. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. IEEE Trans. Softw. Eng. 36, 2 (2010), 226–247.
- M. Harman, Y. Jia, and Y. Zhang. 2015. Achievements, Open Problems and Challenges for Search Based Software Testing. In ICST. 1–12.

Q&A?