

Software Testing 2

Software Engineering
Prof. Maged Elaasar

Learning objectives

- Symbolic execution of code
- Test input generation
- Regression testing
- Other testing kinds (e.g., mutation testing)

Symbolic Execution

Infeasible path

- We are going to perform **symbolic execution** to model individual paths in terms of **logical constraints**.
- For a given decision in the code, we are examining whether there exists an input that evaluates the decision to **true** and whether there exists another input that evaluates it to **false**.
- Paths that cannot be executed under any inputs are called **infeasible paths** or **dead code**.
- This program has only one path, because the decision evaluates always to **true**.

```
int x=0;  
if (x<1) x:=x+1  
else x:=x-1; // dead code
```

Symbolic execution

- Loops must be unrolled first.
- Use fresh variables in the beginning and after each state updates.
- For each decision, we propagate constraints for both **true** and **false** branches.
- For each path, the condition to exercise the path is called the **path condition**.
- The effect of executing statements along a path is called the **path effect**.

Simple example

```
if (x>1) {  
    x := x-2; // stmt1  
} else {  
    x := 2*x; // stmt2  
}  
// stmt3
```

- Path condition for stmt1: $x > 1$
- Path effect at stmt1: $(x > 1) \text{ AND } (x' = x - 2)$
- Path condition for stmt2: $x \leq 1$
- Path effect at stmt2: $(x \leq 1) \text{ AND } (x' = 2x)$
- The overall path effect at stmt3:
 $((x > 1) \text{ AND } (x' = x - 2)) \text{ OR } ((x \leq 1) \text{ AND } (x' = 2x))$

Infeasible path example 1

- Use symbolic execution to model each path in terms of logical conditions.
- What is the number of feasible paths?

```
public static int infeasiblepath(int x) {  
    if (x < 4) {  
        return 0;  
    }  
    int value = 0;  
    int y = 3 * x + 1;  
    if (x * x > y) {  
        value = value + 1;  
    } else {  
        value = value - 1;  
    }  
    return value;  
}
```

Solution: 2 Feasible Paths

- There are 2 decisions in this example
- The first one ($x < 4$) can be evaluated to **true** or **false**.
- The second one ($x * x > y$) cannot go through the **else** branch because when $x \geq 4$, then $x * x > 3x + 1$ is always **true**., i.e., the **else** branch is dead code.
- Since the number of feasible paths should be $2^{\text{(the number of decisions that could evaluate to both **true** or **false**)}}$
- Then the number of feasible paths here is $2^1 = 2$ in this example

Infeasible path example 2

- Use symbolic execution to model each path in terms of logical conditions.
- What is the number of feasible paths?

```
public int pathConstraints(int x) {  
    int value=0;  
    int y=x*x;  
    if (x>3)  
        x = x + 1;  
    else  
        x = abs(x); //abs(x) computes the absolute value of x  
    if (2*x-1>y)  
        y= y * 2;  
    else  
        y = y/ 2;  
    return x * y;  
}
```

Solution: 2 Feasible Paths

- Path1 (TT): $x > 3$ AND $2(x+1)-1 > x^2$
 $\Rightarrow x > 3$ AND $2x+1 > x^2$ (**NOT FEASIBLE**)
- Path2 (TF): $x > 3$ AND $2(x+1)-1 \leq x^2$
 $\Rightarrow x > 3$ AND $2x-1 \leq x^2$
(**FEASIBLE**, for example when $x=4$)
- Path3 (FT): $x \leq 3$ AND $2\text{abs}(x)-1 > x^2$
 $\Rightarrow x \leq 3$ AND $((x \geq 0 \text{ AND } 2x-1 > x^2) \text{ OR } (x < 0 \text{ AND } -2x-1 > x^2))$
 $\Rightarrow (x \leq 3 \text{ AND } x \geq 0 \text{ AND } 2x-1 > x^2) \text{ OR } (x \leq 3 \text{ AND } x < 0 \text{ AND } -2x-1 > x^2)$
 $\Rightarrow (0 \leq x \leq 3 \text{ AND } 2x-1 > x^2) \text{ OR } (x < 0 \text{ AND } -2x-1 > x^2)$ (**NOT FEASIBLE**)
- Path4 (FF): $x \leq 3$ AND $2\text{abs}(x)-1 \leq x^2$
 $\Rightarrow x \leq 3$ AND $((x \geq 0 \text{ AND } 2x-1 \leq x^2) \text{ OR } (x < 0 \text{ AND } -2x-1 \leq x^2))$
 $\Rightarrow (x \leq 3 \text{ AND } x \geq 0 \text{ AND } 2x-1 \leq x^2) \text{ OR } (x \leq 3 \text{ AND } x < 0 \text{ AND } -2x-1 \leq x^2)$
 $\Rightarrow (0 \leq x \leq 3 \text{ AND } 2x-1 \leq x^2) \text{ OR } (x < 0 \text{ AND } -2x-1 \leq x^2)$
(**FEASIBLE**, for example $x=3 \Rightarrow 2x-1 \leq x^2 \Rightarrow 5 \leq 9$)

```
public int pathConstraints(int x) {  
    int value=0;  
    int y=x*x;  
    if(x>3)  
        x = x + 1;  
    else  
        x = abs(x);  
    if(2*x-1>y)  
        y= y * 2;  
    else  
        y = y/ 2;  
    return x * y;  
}
```

Test Input Generation

How to generate test inputs?

- You can use symbolic execution to generate test inputs
- If you want to exercise a particular path, first determine its path condition.
- Find concrete input values that make the path condition true

Simple example

```
if (x>1) {  
    x= x-2; //stmt1  
} else {  
    x= 2x; //stmt2  
}  
// stmt3
```

One decision means $2^1 = 2$ paths

Path conditions \Rightarrow concrete input

Path (T): $x > 1 \Rightarrow T(x=2)$

Path (F): $x \leq 1 \Rightarrow T(x=1)$

Test generation example 1

1. Use symbolic execution to model each path in terms of logical constraints.
2. Find concrete input assignments for each path condition

```
public static int generate_test_for_this1(int x, int y, int z){  
    if(x < y)  
        z++;  
    else  
        z--;  
    if(z < 2*x+5)  
        x++;  
    else  
        y++;  
    return y;  
}
```

Solution

```
if (x < y) z++; else z--;  
if (z < 2*x+5) x++; else y++;
```

Two decisions means $2^2 = 4$ paths

Path conditions \Rightarrow concrete input

Path (TT): $x < y$ AND $z+1 < 2x+5 \Rightarrow T(x=1, y=2, z=5)$

Path (TF): $x < y$ AND $z+1 \geq 2x+5 \Rightarrow T(x=1, y=2, z=7)$

Path (FT): $x \geq y$ AND $z-1 < 2x+5 \Rightarrow T(x=1, y=1, z=5)$

Path (FF): $x \geq y$ AND $z-1 \geq 2x+5 \Rightarrow T(x=1, y=1, z=8)$

Test generation example 2

1. Use symbolic execution to model each path in terms of logical constraints.
2. Find concrete input assignments for each path condition

```
public static int generate_test_for_this2(int x, int y, int z) {  
    for(int i=0; i<2; i++) {  
        if(x < y)  
            Z++;  
        else  
            Z--;  
        if(z < 2*x+5)  
            X++;  
        else  
            Y++;  
        return y;  
    }  
}
```


Solution

- The same program (as example 1) but with a bounded loop

```
for (int i=0; i<2; i++) {  
    if (x<y)    z++; else z--;  
    if (z<2*x+5) x++; else y++;  
}
```

- Perform loop unrolling :

D1: **if** (x<y) z++; **else** z--;

D2: **if** (z<2*x+5) x++; **else** y++;

D3: **if** (x<y) z++; **else** z--;

D4: **if** (z<2*x+5) x++; **else** y++;

- 4 decisions means $2^4 = 16$ paths

Solution (cont'd)

- Path Conditions for each path

TTTT: $x < y$ AND $z+1 < 2x+5$ AND $x+1 < y$ AND $z+2 < 2(x+1)+5$

TTTF: $x < y$ AND $z+1 < 2x+5$ AND $x+1 < y$ AND $z+2 \geq 2(x+1)+5$

TTFT: $x < y$ AND $z+1 < 2x+5$ AND $x+1 \geq y$ AND $z < 2(x+1)+5$

TTFF: $x < y$ AND $z+1 < 2x+5$ AND $x+1 \geq y$ AND $z \geq 2(x+1)+5$

TFTT: $x < y$ AND $z+1 \geq 2x+5$ AND $x < y+1$ AND $z+2 < 2(x+1)+5$

TFTF: ... (Continue in a similar manner. Note that later decisions are affected by prior assignments. This case is for demonstration purposes only and actual questions will not ask you to list all 16 different path conditions since it is tedious)

D1: `if (x<y) z++; else z--;`

D2: `if (z<2*x+5) x++; else y++;`

D3: `if (x<y) z++; else z--;`

D4: `if (z<2*x+5) x++; else y++;`

Software Testing 3 Quiz

Regression Testing

Regression testing (retesting)

- Suppose that you've tested a product thoroughly and found no errors.
- Suppose that the product is then changed in one area and you want to be sure that it still passes all the tests it did before the change.
- Regression testing is one designed to make sure that the software hasn't regressed (or taken a step backward).
- The main idea is to select the minimum set of tests for possible regression

Regression Test Selection (RTS)

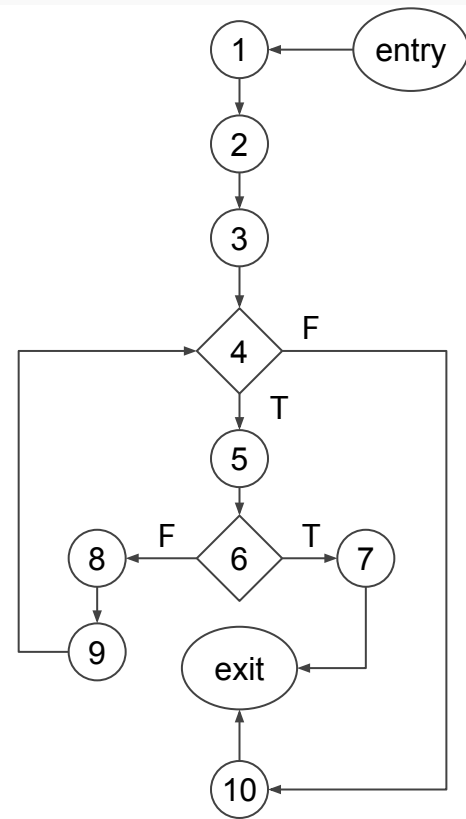
- P: old version
- P': new version
- T is a test suite for P
- Assume that all tests in T ran on P \Rightarrow generate coverage matrix C.
- Given the delta between P and P' and the coverage matrix C, identify a subset of T that can identify all regression faults \Rightarrow Safe **RTS** (regression test selection)

Harrold & Rothermel's RTS

- A safe, efficient regression test selection technique
- Regression test selection based on traversal of control flow graphs for the old and new version.
- The key idea is to select tests that will exercise **dangerous edges** in the new program version.
- Dangerous edges are edges in the old CFG whose target nodes are different in the new CFG
- If you find the subset of test cases that exercised dangerous edges, this subset is as effective as running all test cases.

Step 1: build the CFG of the original program

```
1  public static int avg(File file, int max) {  
2      int[] nums = new int[max];  
3      int count = 0;  
4      Scanner scanner = new Scanner(file));  
5      while(scanner.hasNextInt()) {  
6          int n = scanner.nextInt();  
7          if (n < 0) {  
8              return ERROR;  
9          } else {  
10             nums[count] = n;  
11             count++;  
12         }  
13     }  
14     return calcAvg(nums);  
15 }
```



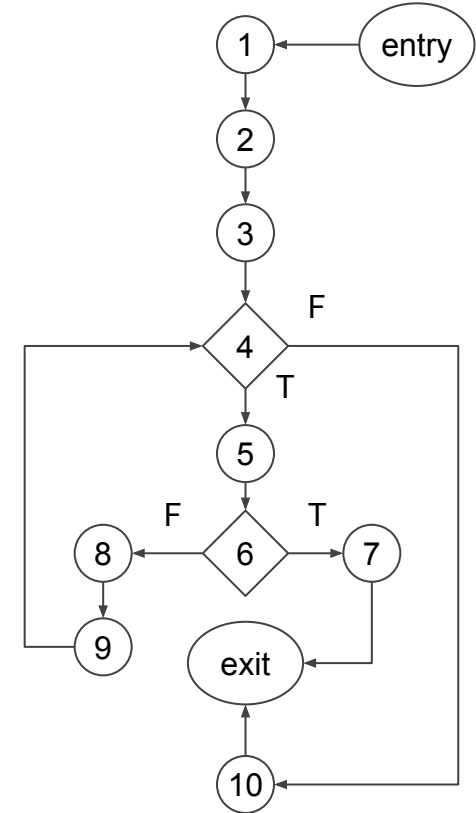
Step 2: run $T = \{T1, T2, \dots\}$ on P

Test	Input	Output	Edges Traversed
T1	{}, 100	0	entry, 1, 2, 3, 4, 10, exit
T2	{-1}, 100	ERROR	entry, 1, 2, 3, 4, 5, 6, 7, exit
T3	{1,2,3}, 100	2	entry, 1, 2, 3, 4, 5, 6, 8, 9, 4, 5, 6, 8, 9, 4, 5, 6, 8, 9, 4, 10, exit

```
public static int avg(File file, int max){
1    int[] nums = new int[max];
2    int count = 0;
3    Scanner scanner = new Scanner(file));
4    while(scanner.hasNextInt()) {
5        int n = scanner.nextInt();
6        if (n < 0) {
7            return ERROR;
8        } else {
9            nums[count] = n;
10           count++;
11       }
12   }
13   return calcAvg(nums);
14 }
```

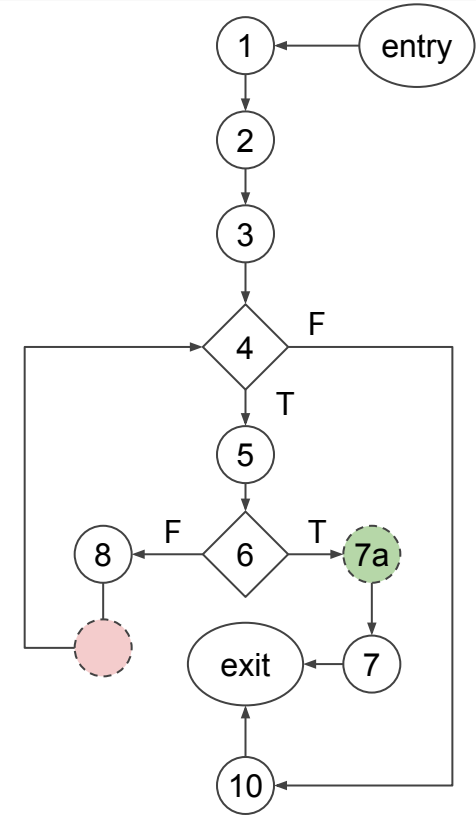
Step 3: build edge coverage matrix

Edge	T1	T2	T3
(entry, 1)	1	1	1
(1, 2)	1	1	1
(2, 3)	1	1	1
(3, 4)	1	1	1
(4, 5)	0	1	1
(5, 6)	0	1	1
(6, 7)	0	1	0
(7, exit)	0	1	0
(4, 10)	1	0	1
(10, exit)	1	0	1
(6, 8)	0	0	1
(8, 9)	0	0	1
(9, 4)	0	0	1



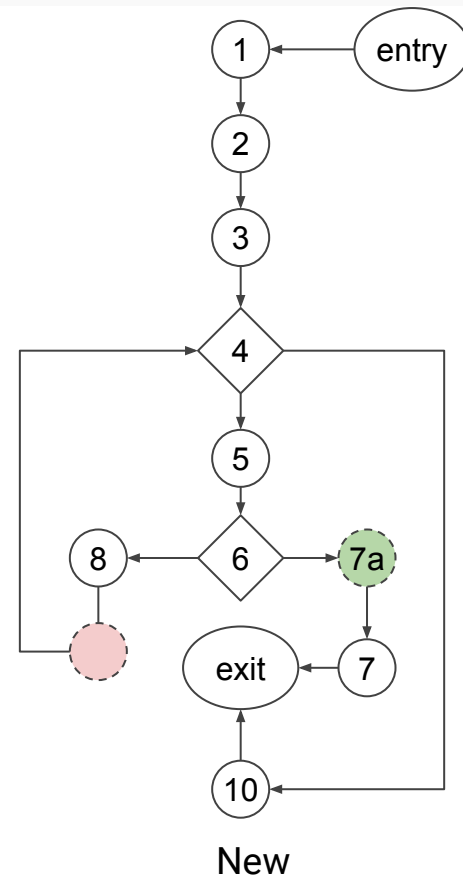
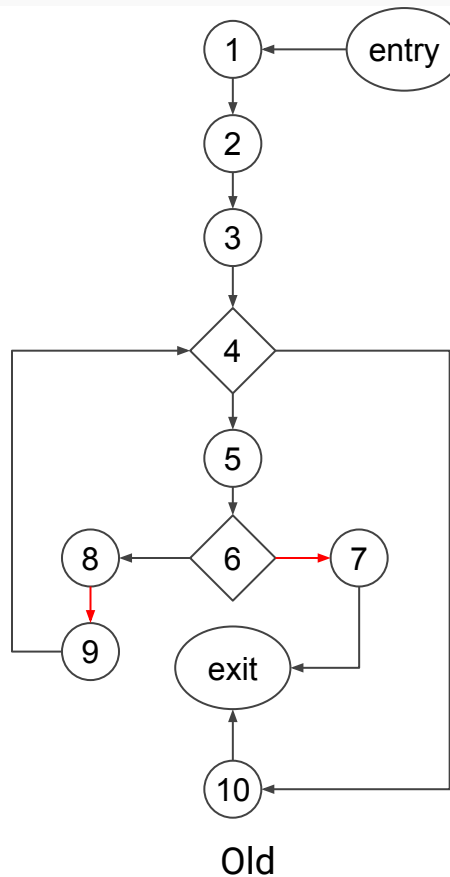
Step 4: build the CFG of the changed code

```
public static int avg(File file, int max) {  
1   int[] nums = new int[max];  
2   int count = 0;  
3   Scanner scanner = new Scanner(file);  
4   while(scanner.hasNextInt()) {  
5       int n = scanner.nextInt();  
6       if (n < 0) {  
7a      System.out.print("bad input");  
7       return ERROR;  
8       } else {  
9           nums[count] = n;  
10          count++;  
11      }  
12  }  
13  return calcAvg(nums);  
14  }
```



Step 5: traverse the two CFGs in parallel

- Identify dangerous edges which are edges in the old CFG where their target nodes are different in the new CFG
- In this case:
 - (6, 7)
 - (8, 9)



Step 6: select tests covering dangerous edges

Edge	T1	T2	T3
(entry, 1)	1	1	1
(1, 2)	1	1	1
(2, 3)	1	1	1
(3, 4)	1	1	1
(4, 5)	0	1	1
(5, 6)	0	1	1
(6, 7)	0	1	0
(7, exit)	0	1	0
(4, 10)	1	0	1
(10, exit)	1	0	1
(6, 8)	0	0	1
(8, 9)	0	0	1
(9, 4)	0	0	1

- Which tests are relevant to changes and thus must be rerun?
- In this case:
 - T2, T3

RTS practice question (at home)

1. Draw the control flow graphs for the old and new versions of the program
2. Mark the dangerous edges on your control flow graphs.

```
1  void fun(int n, int k){  
2      int sum = 0;  
3      int product = 1;  
4      for(i = 1; i <= n; i++){  
5          sum = sum + i;  
6          if (k%2==0)  
7              product = product*i;  
8      }  
9      write(sum);  
10     write(product);  
11 }
```

```
1  void fun(int n, int k){  
2      int sum = 0;  
3      int product = 1;  
4      for(i = 1; i <= n; i++){  
5          sum = sum + i;  
6          if (k%2==0)  
7              product = product*2*i;  
8      }  
9      write(sum);  
10     write(product);  
11 }
```

RTS practice question (cont'd)

3. Identify a subset of the following tests that are relevant to the edits and thus must be re-run for the new version of the program.

T1 ($n=0, k=1$)

T2 ($n=10, k=2$)

T3 ($n=1, k=3$)

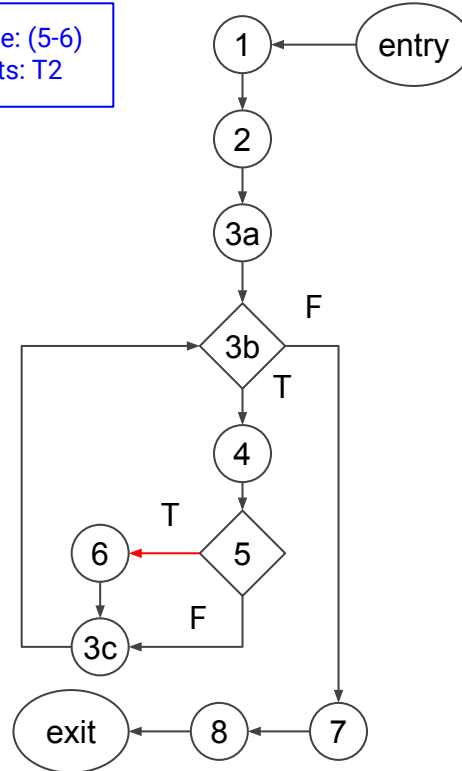
Work out the steps at home

The answer should be that T2 needs to be rerun for the new version.

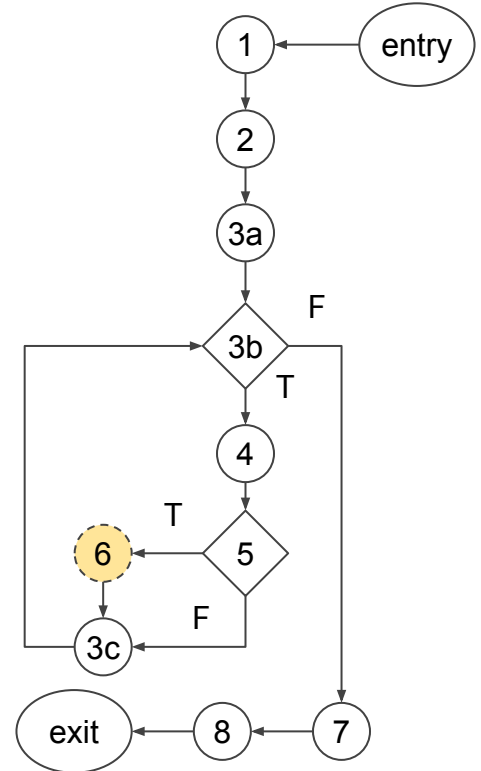
RTS practice question - solution

Edge	T1	T2	T3
(entry, 1)	1	1	1
(1, 2)	1	1	1
(2, 3a)	1	1	1
(3a, 3b)	1	1	1
(3b, 4)	0	1	1
(3b, 7)	1	1	1
(3c, 3b)	0	1	1
(4, 5)	0	1	1
(5, 6)	0	1	0
(5, 3c)	0	0	1
(6, 3c)	0	1	0
(7, 8)	1	1	1
(8, exit)	1	1	1

Dangerous edge: (5-6)
Regression tests: T2



Old



New

Harrold et al. RTS for Java

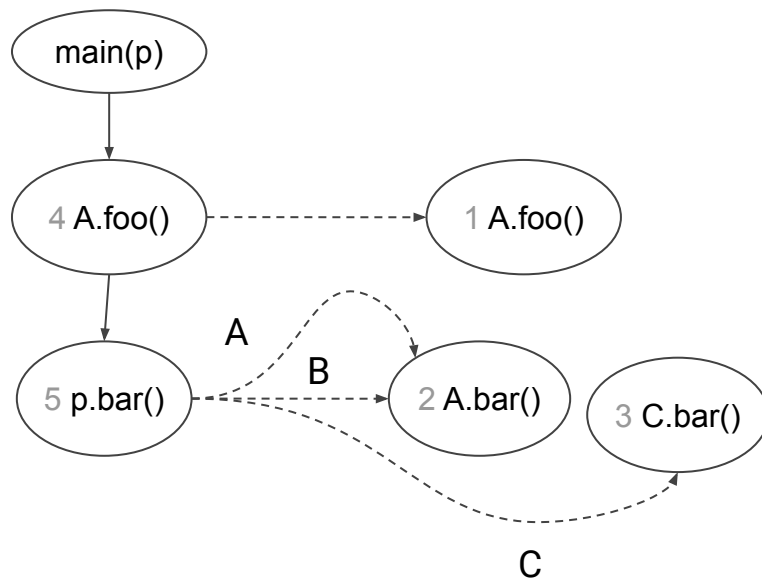
- What is one of the main challenges for making RTS work in Java?
- Polymorphism (dynamic binding) allows a member function call to be resolved at runtime

```
class A {  
    public static void foo() { ... }  
    public void bar() { ... }  
}  
class B extends A {  
}  
class C extends B {  
    public void bar() { ... }  
}  
void main (A p) {  
    A.foo(); // static method call  
    p.bar(); // which version of bar() will be called?  
}
```

JIG (Java Interclass Graph)

- JIG extends CFG to handle polymorphic method invocations

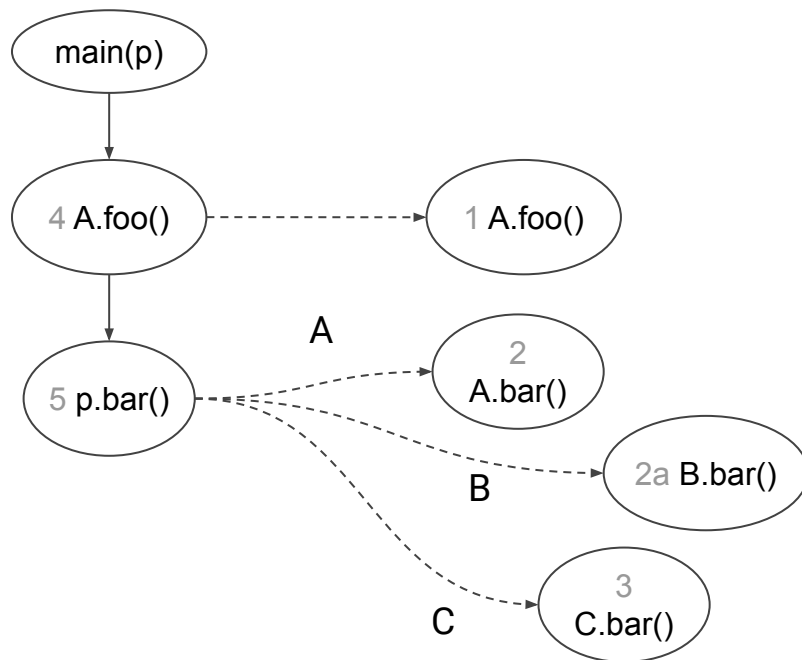
```
class A {  
1   public static void foo() { ... }  
2   public void bar() { ... }  
}  
class B extends A {  
}  
class C extends B {  
3   public void bar() { ... }  
}  
void main (A p) {  
4   A.foo();  
5   p.bar();  
}
```



New JIG when the code changes

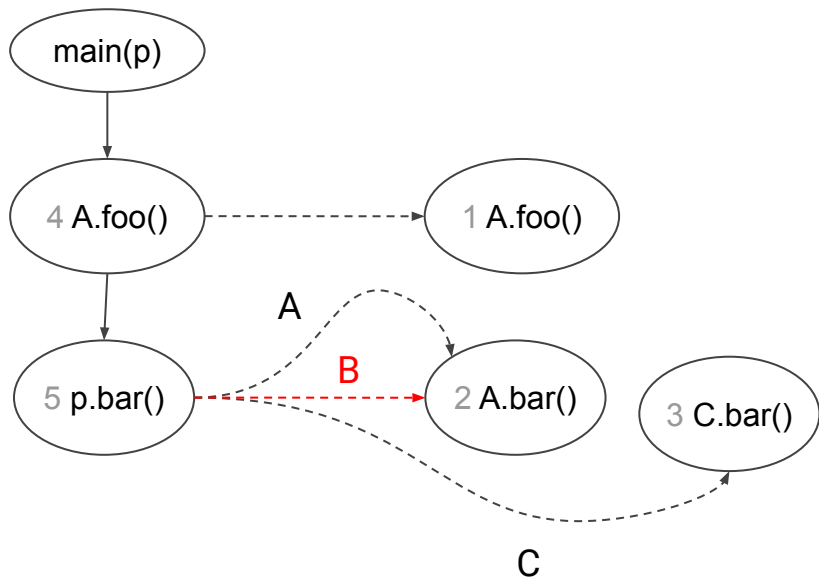
- When the code changes, draw the new JIG

```
class A {  
1   public static void foo() { ... }  
2   public void bar() { ... }  
}  
class B extends A {  
2a  public void bar() { ... }  
}  
class C extends B {  
3   public void bar() { ... }  
}  
void main (A p) {  
4   A.foo();  
5   p.bar();  
}
```

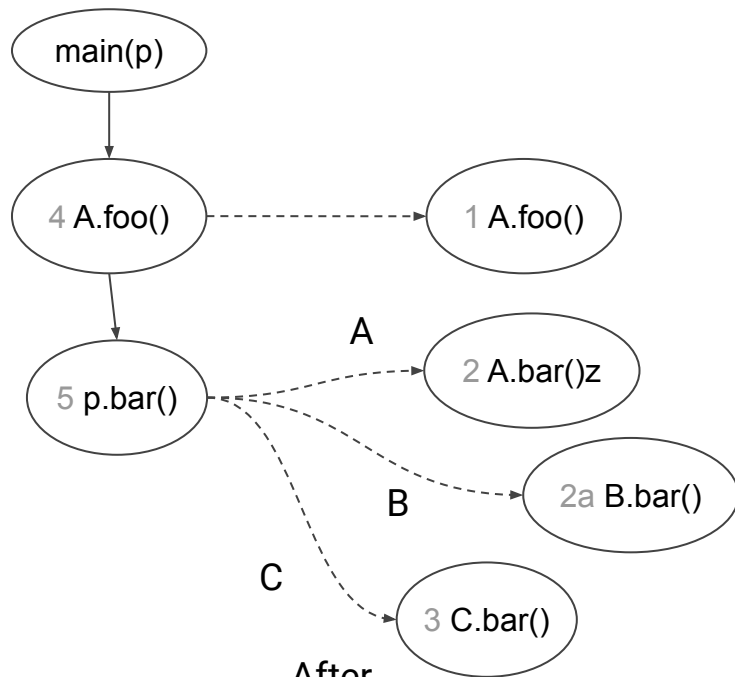


Identify the dangerous edges

- Compare the two JIGs and identify the dangerous edges
 - Calling p.bar() on object of type B in this case



Before

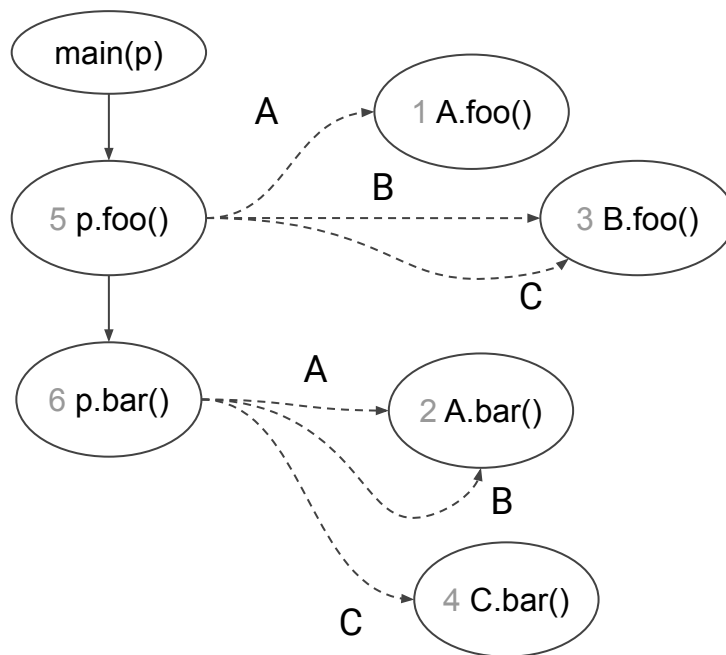


After

Another example

- Draw the JIG of the following code

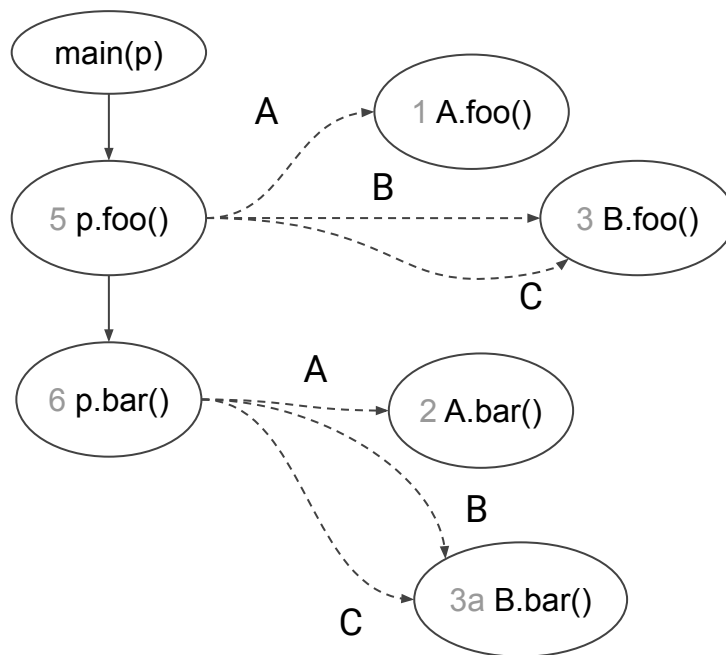
```
1  class A {  
2      public void foo() { ... }  
3      public void bar() { ... }  
4  }  
5  class B extends A {  
6      public void foo() { ... }  
7  }  
8  class C extends B {  
9      public void bar() { ... }  
10 }  
11 void main (A p) {  
12     p.foo();  
13     p.bar();  
14 }
```



Another example

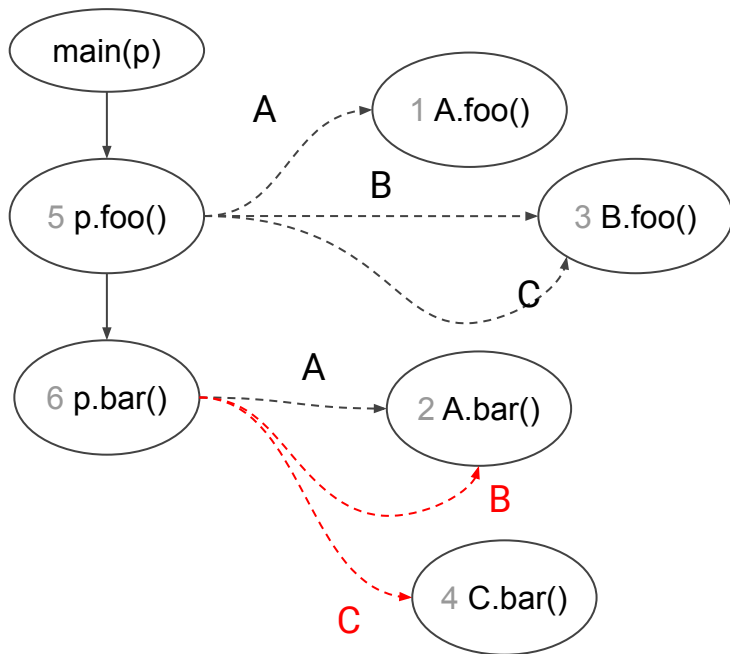
- Draw the JIG of the changed code

```
class A {  
1   public void foo() { ... }  
2   public void bar() { ... }  
}  
class B extends A {  
3   public void foo() { ... }  
3a  public void bar() { ... }  
}  
class C extends B {  
   public void bar() { ... }  
}  
void main (A p) {  
5   p.foo();  
6   p.bar();  
}
```

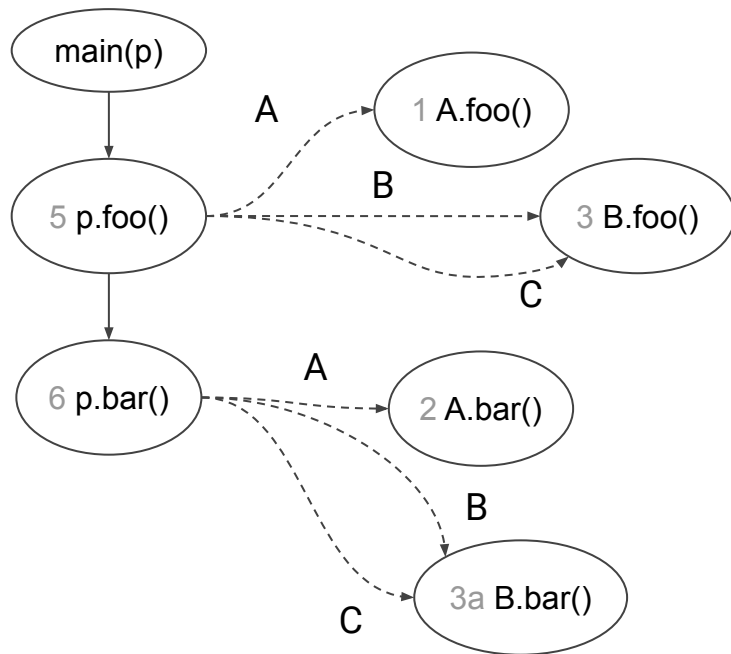


Identify dangerous edges

- Compare the JIGs to identify the dangerous edges
 - In this case, it is calling p.bar() on objects B and C



Before



After

Other Testing Methods

What other testing methods exist?

- **Data flow testing:** exploring the paths between data definition and data use
- **Randomized testing:** choosing samples from the input space randomly
 - Does not guarantee that the test coverage will improve
 - Can help systematic testing by exploring variations but not a substitute
- **Model-based (black-box) testing:** enumerate test cases based on specifications and abstract models (e.g., UML behavior diagrams, grammar production rules)
- **Mutation testing:** mutating the code to test the adequacy of the test suite in discovering defects

Mutation testing

- The goal of mutation testing is to assess or improve the efficacy of test suites in discovering defects.
- The process, given program P and test suite T , is as follows:
 - We systematically apply mutations to the program P to obtain a sequence P_1, P_2, \dots, P_n of mutants of P . Each mutant is derived by applying a single mutation operation to P .
 - We run the test suite T on each mutant; T is said to kill mutant P_j if it detects an error.
 - If it kills k out of n mutants, the adequacy of T is measured by the quotient k/n , which is also called its **mutant killing ratio**. T is mutation adequate if $k=n$.
- Three kinds of mutations exist:
 - **Value mutations**: changes to value of constants, parameters, or loop bounds
 - **Decision mutations**: changes conditions to reflect common slips and errors
 - **Statement mutations**: includes deleting lines, swapping their order, change math operators

Java mutation operators

Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	<i>super</i> keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	<i>new</i> method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	<i>this</i> keyword deletion
	JSC	<i>static</i> modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Value mutation example

```
/*  
 * counts the number of elements t[i] that are  $l < t[i] < u$   
 * t[] is assumed to be in ascending order and  $l < u$   
 */
```

```
public int segment (int t[], int l, int u) {
```

```
    int k = 0;
```

Mutating to k=1 causes miscounting

```
    for(int i=0; i<t.length; i++) {
```

```
        if (t[i]>l && t[i]<u) {
```

```
            k++;
```

```
        }
```

```
    }
```

```
}
```

Mutating to i=1 causes miscounting

T({1, 2, 3}, 2, 4) can kill this mutant

T({2}, 1, 10) can kill this mutant

Decision mutation example

```
/*  
 * counts the number of elements t[i] that are  $l < t[i] < u$   
 * t[] is assumed to be in ascending order and  $l < u$   
 */
```

```
public int segment (int t[], int l, int u) {
```

```
    int k = 0;
```

Mutating to $i \leq t.length$ causes miscounting

```
    for(int i=0; i<t.length; i++) {
```

```
        if (t[i]>l && t[i]<u) {
```

```
            k++;
```

```
        }
```

```
    }
```

```
}
```


Mutating to $t[i] > u$ causes miscounting

$T(\{2\}, 1, 10)$ can kill this mutant

Any test can kill this mutant.
Why?

Statement mutation example

```
/*  
 * counts the number of elements t[i] that are l<t[i]<u  
 * t[] is assumed to be in ascending order and l<u  
 */  
public int segment (int t[], int l, int u) {  
    int k = 0;  
  
    for(int i=0; i<t.length; i++) {  
        if (t[i]>l && t[i]<u) {  
            k++;  
        }  
    }  
}
```



Mutating by removing or duplicating causes miscounting

T({2}, 1, 10) can kill
both mutants

Software Testing 4 Quiz

References

- Rothermel, G., Harrold, M. J.: “A safe, efficient regression test selection technique.” ACM Transactions on Software Engineering and Methodology, 6(2):173–210, Apr. 1997.
- G., Harrold, et. al: "Regression Test Selection for Java Software." Proc. of OOPSLA, 2001.
- Further reading (for those interested):
 - <http://ix.cs.uoregon.edu/~michal/book/Samples/book-ch14-structural.pdf>
 - <http://season-lab.github.io/papers/survey-symbolic-execution-preprint-CSUR18.pdf>