CS 130 SOFTWARE ENGINEERING

TESTING

:BOUNDED ITERATION INFEASIBLE PATHS

Professor Miryung Kim
UCLA Computer Science
Based on Materials from Miryung Kim

AGENDA

- Part 1: Testing adequacy criteria and Junit
- **Part 2: The number of paths and Infeasible paths**
- **Part 3: Symbolic execution Test Generation**
- **Part 4: Regression test selection**

MORE SPECIFICALLY

- Counting the number of loop iterations for bounded programs
- Identify infeasible paths through symbolic execution

ESTIMATING THE NUMBER OF PATHS

ESTIMATING THE NUMBER OF PATHS

- For a loop-free program with k branches, the number of paths is 2^k if no infeasible path exists.
- How about a program with loops?
- In this exercise, we perform loop unrolling then the number of paths is 2^(# branches) when each branch can be executed both true or false.

EXAMPLE 1. ESTIMATING LOOP ITERATIONS

```
public static int fun1(int N) {

    int sum = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= Math.pow(3, i); j+
+) {
            System.out.println("HelloWorld");
            if (new Random().nextInt() % 2 == 0)
                sum++;
        }
    }
    return sum;
}
```

EXAMPLE 1. ESTIMATING LOOP ITERATIONS

- 1. What is the number of times that "HelloWorld" will be printed?
- 2. What is the number of branch executions in the loop-unrolled program?
- 3. What is the number of paths?

EXAMPLE 2:

```
public static void fun2(int N) {
        int sum = 0;
        Random rand = new Random();
        for ( int i = 1; i <= N; i ++ ){
           for( int j = 1; j <= Math.pow(3,i); j ++){
              if (rand.nextInt() %2 ==0)
                 sum++;
           }
           for (int k=1; k<=i; k++) {
              if (Math.pow(2,k) % 2 ==0) sum++;
           }
        }
        System.out.println("N:"+N+ "\tSum:" +sum);
   }
```

EXAMPLE 2

- 1. What is the number of branch executions in the loop-unrolled program?
- 2. What is the number of paths?

ABOUT LOOP J'S ITERATION

When i is 1, the loop j executes 3^1.
When i is 2, the loop j executes 3^2.
...
When i is N, the loop j executes 3^N.

T1(N)=3^1 + 3^2 + ... 3^N
3T1(N)=   3^2 + ... 3^N +3^(N+1)
Subtract the first from the second

2T1(N) = 3^(N+1) - 3
T1(N) = (3^(N+1) - 3)/2

ABOUT LOOP K'S ITERATION

When i is 1, the loop j executes 1.
When i is 2, the loop j executes 2.
...
When i is N, the loop j executes N.

T1(N)=1 + 2 + ... N
T1(N)=N + N-1 + ... 1

2T1(N) = (1+N) *(N) // (1+N) appear N times

T1(N) = N(N+1)/2

- rand.nextInt()%2==0 can evaluate true or false randomly.

- Math.pow(2,k)%2==0 is always true because (2^k)%2 is always 0. So this branch is deterministic regardless of inputs provided to the program.

- If the second branch was the same as the first, what will the answer be?

PRACTICE QUESTION

```
sum = 0;
for (int i = 0; i < N; i++ ){
            for (int j = 1; j <= pow(2,i); j++){
            if (a[i]<k)
            sum++;}
}
for (int m=1; m<=i; k++) {
            if (pow(2,m)  % 2 == 1) { sum++;} // false
      else {
           sum--;}
} }
```

The second branch always evaluates to false for every possible input. So the branch (pow(2,m)%2 does not contribute to increasing the number of path).

For the first branch

When i=0, the inner j loop iterates 2^0

When i=1, the inner j loop iterates 2^1

...

When i=n, the inner j loop iterates 2^n,

Let's call T(n) the total number of times that the inner loop j iterates.

T(n) = 2^0 + 2^1 + .. 2^n

Multiply T(n) by 2 on both sides of the equation

2T(n) = 2^1 + 2^1 + .. 2^(n+1)

Subtract the first equation from the second,

T(n) = 2^(n+1) - 2^0

T(n) = 2^(n+1) -1

The branch (a[i]<k) could evaluate to TRUE for some input and FALSE for some other input, however it evaluates to the same value within each i.

So the total number of paths is 2^(n), not 2^(n+1)-1).

INFEASIBLE PATH

- Paths that cannot be executed under any inputs are called “infeasible paths” commonly called as, dead code.
- We are going to teach a symbolic execution to model individual paths in terms of **logical constraints.**

- In other words, for a given branch b, we are examining whether there exists an input that evaluate the branch to be true and whether there exists another input that evaluates the branch to be false.
- This program has only one path, because the branch execution is always FALSE.

```
int x=0;
if (x>1) x:=x+1 else x:=x-1;
```

SYMBOLIC EXECUTION

SYMBOLIC EXECUTION

- Use fresh variables in the beginning
- Use fresh variables after the state updates
- Loop must be unrolled first
- For each branch, we propagate the constraints for both true and false evaluation of the branch
- Basically, we are conservatively estimate the effect of taking either path.

PATH CONDITION AND EFFECT

- For each path, the condition to exercise the path is called a "path condition."
- The effect of executing statements along a possible path is called "effect"

SIMPLE EXAMPLE

```
if (x>1) {
     x= x-2;  //stmt 1
}else {
     x= 2*x;  //stmt 2
}
// stmt 3
```

Path condition for stmt1 is x>1

Path condition for stmt2 is x<=1

Effect for stmt1 (x>1) AND (x’= x-2)

Effect for stmt2 (x<=1) AND (x’= 2x)

The overall symbolic execution result at stmt3

((x>1) AND (x`=x-2)) OR ((x<=1) AND (x’=2x))

INFEASIBLE PATH EXAMPLE 1.

```
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;
```

INFEASIBLE PATH EXAMPLE 1.

- 1. Draw a control flow graph for the program.
- 2. What is the number of paths?
- 3. Use symbolic execution to model each path in terms of logical constraints.

- There is one path that returns immediate when $x<4$.
- $x*x>y$ cannot go through "else" side because when $x>=4$, $x*x>3x+1$ is always true.
- The else side is dead code.

- Going back to the concept of the number of feasible paths, the paths is 2^(the number of branch executions that could evaluate to both TRUE for some input or FALSE for some other input).
- In other words, for a given branch b, we are examining whether there exists an input that evaluate the branch to be true and whether there exists another input that evaluates the branch to be false.

PRACTICE ON INFEASIBLE PATH

```
public int pathConstraints(int x){
    int value = 0;
    int y=x*x;
    if(x>3) x = x + 1; else x = abs(x)
;// assume that abs(x) computes the
absolute value of a.
    if(2*x-1>y) y= y * 2; else y = y/
2;
    System.out.print(“130 is a
software engineering course.”);
```

SOLUTION

Path1 TT: x>3 AND 2(x+1)-1>x^2

== x>3 AND 2x+1>x^2 (NOT FEASIBLE)

Path2 TF: x>3 AND 2(x+1)-1<=x^2

x>3 AND 2x-1<=x^2 (FEASIBLE, when x=3)

Path3 FT: x<=3 AND 2(abs(x))-1>x^2

x<=3 AND (x>=0 AND 2x-1>x^2 OR (x<0 AND -2x-1>x^2))

==0<=x<=3 AND 2x-1>x^2 OR (x<0 AND 0>x^2+2x+1)) NOT FEASIBLE

Path4 FF: x<=3 AND 2abs(x)-1<=x^2

x<=3 AND (x>=0 AND 2x-1<=x^2) OR (x<0 AND -2x-1<=x^2)
(FEASIBLE, for example x=3, 2x-1<x^2, 5<9)

TEST INPUT GENERATION

HOW TO GENERATE TEST INPUTS?

- You can use symbolic execution to generate concrete inputs
- If you want to exercise a particular path, first determine a path condition.
- Find concrete input assignments for each path

SIMPLE EXAMPLE

```
if (x>1) {
      x= x-2;  //stmt 1
}else {
      x= 2x;  //stmt 2
}
// stmt 3
```

Path condition for stmt1 is x>1

Path condition for stmt2 is x<=1

Effect for stmt1 (x>1) AND (x'= x-2)

Effect for stmt2 (x<=1) AND (x'=2x)

The overall symbolic execution result at stmt3

((x>1) AND (x`=x-2)) OR ((x<=1) AND (x'=2x))

⇒ The test input you need is any x where x>1 and any x where x<=1.

⇒ Concretely x=2 exercises stmt 1 and x=1 exercises stmt 2

TEST GENERATION: #1

```
public static int generate_tests_for_this1(int x,
int y, int z) {
     if (x < y) {
        z++;
     } else {
        z--;
     }
     if (z < 2 * x + 5) {
        x++;
     } else {
        y++;
     }
     return y;
}
```

TEST GENERATION: #1

- 1. Draw a control flow graph for the program.
- 2. Use symbolic execution to model each path in terms of logical constraints.
- 3. Find concrete input assignments for each path condition

- `if (x<y)  z++; else z--;`
- `if (z<2*x+5) x++; else y++;`
- Path Conditions
- TT: x<y AND z+1 <2x+5
- TF: x<y AND z+1 >=2x+5
- FT: x>=y AND z-1 <2x+5
- FF: x>=y AND z-1 <2x+5

TEST GENERATION #2

```
public static int generate_tests_for_this2(int x,
int y, int z) {
     // a bit time consuming, so we will do this
question if we have a time,
     // otherwise complete it at home.
     for (int i = 0; i < 2; i++) {
        if (x < y) {
           z++;
        } else {
           z--;
        }
        if (z < 2 * x + 5) {
           x++;
        } else {
           y++;
        }
     }
     return y;
```

- Q2. The same program with a finite loop
- for (int i=0; i<2; i++) {
-         if (x<y) z++; else z--;
-         if (z<2*x+5) x++; else y++;
- }

- Loop Unrolling
- B1:   if (x<y) z++; else z--;
- B2:   if (z<2*x+5) x++; else y++;
- B3:   if (x<y) z++; else z--;
- B4:   if (z<2*x+5) x++; else y++;

- 4 branch executions => 16 paths

- 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 >= 2(x+1)+5
- TTFT: x<y AND z+1 <2x+5 AND x+1 >= y AND z < 2(x+1)+5
- TTFF: x<y AND z+1 <2x+5 AND x+1 >= y AND z >= 2(x+1)+5
- TFTT: x<y AND z+1 >=2x+5 AND x <y +1 AND z+2 < 2(x+1)+5
- TFTF: ... (Continue in a similar manner. Note that later branch conditions are affected by prior assignments. This case is for demonstration purposes and the exam question will not ask you to list 16 different path conditions tediously, but may require understanding the concept of path conditions.)
- TFFT:
- TFFF:
- FTTT:
- FTTF:
- FTFT:
- FTFF:
- FFTT:
- FFTF:
- FFFT:
- FFFF:

EQUIVALENCE PARTITIONING

- A good test case covers a large part of the possible input data.
- The concept of "equivalence partitioning" is a formalization of this idea and helps reduce the number of test cases required.

RECAP.

- We have studied three different coverage metrics.
- We have studied how to count the number of paths when the number of loops is bounded.

PREVIEW

- We will discuss regression test selection, prioritization, and augmentation methods.

REGRESSION TESTING

AGENDA

- Regression testing selection
- Change impact analysis: which tests are affected by program changes?

REGRESSION TESTING

- Running tests again to ensure that code changes are not degrading correctness properties
-

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.
- Testing designed to make sure that the software hasn’t taken a step backward, or “regressed” is called “regression testing”

WHAT IS REGRESSION TESTING?

- Regression testing is performed on modified software to provide confidence that
- software behaves correctly and
- modifications did not adversely impact software quality.

REGRESSION TESTING

- Test Case (t)
- e.g. JUnit test
- Test suite: a set of test cases, T= {t1, t2, t3, … tn}
- Regression testing intends to identify regression fault introduced due to changes.
- Regression test strategy?
- The most naive one is to rerun every test case in the test suite.

REGRESSION TEST SELECTION

- P: old version
- P’: new version
- T is a test suite for P
- Assume that all tests in T ran on P. => 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. (Safe RTS)
-

REGRESSION TEST PRIORITIZATION

- P: old version
- P': new version
- T is a test suite for P
- Assume that programmers do not have enough time to select and run test cases.
- How can we order and rank test cases so that test cases that run early can provide the most benefit when the time is limited?
- Given the delta between P and P' and C, what is an ordering of test cases in T?

REGRESSION TEST AUGMENTATION

- P: old version
- P’: new version
- T is a test suite for P
- Generate a set of test cases that effectively exercise the delta between P and P’.
- In other words, it is a test generation for evolving programs.

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.

HARROLD & ROTHERMEL'S RTS

- Dangerous edges are the edges where target node is different

STEP 1. BUILD CFG

Control flow graph for the old version

```
     Procedure avg

S1.  count = 0
S2.  fread(fileptr,n)
P3.  while (not EOF) do
P4.     if (n<0)
S5.        return(error)
        else
S6.        numarray[count] = n
S7.        count++
        endif
S8.     fread(fileptr,n)
     endwhile
S9.  avg = calcavg(numarray,count)
S10. return(avg)
```

STEP 2. RUN T = {T1, T2, ...} ON P

Test Information			
Test	Type	Output	Edges Traversed
t1	Empty File	0	
t2	−1	Error	
t3	1 2 3	2	

STEP 3. BUILD EDGE COVERAGE MATRIX

Test History

Edge	TestsOnEdge(edge)
(entry, D)	111
(D, S1)	111
(S1, S2)	111
(S2, P3)	111
(P3, P4)	011
(P3, S9)	101
(P4, S5)	010
(P4, S6)	001
(S5, exit)	010
(S6, S7)	001
(S7, S8)	001
(S8, P3)	001
(S9, S10)	101
(S10, exit)	101

STEP 4. TRAVERSE TWO CFGS IN PARALLEL

Control flow graph for the old version

Procedure avg2

```
S1'. count = 0
S2'. fread(fileptr,n)
P3'. while (not EOF) do
P4'.    if (n<0)
S5a.       print("bad input")
S5'.       return(error)
        else
S6'.       numarray[count] = n

         endif
S8'.     fread(fileptr,n)
      endwhile
S9'. avg = calcavg(numarray,count)
S10'.return(avg)
```

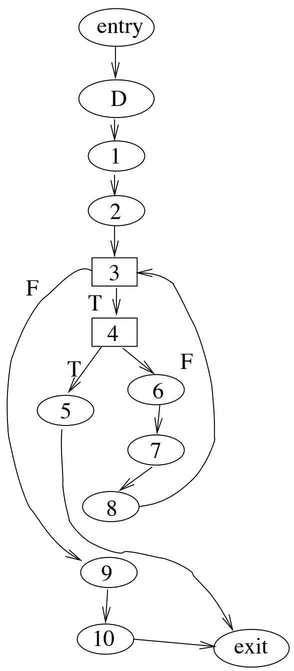
STEP 5. SELECT TESTS THAT ARE RELEVANT TO DANGEROUS EDGES

- Which tests are relevant to changes and thus must be rerun?

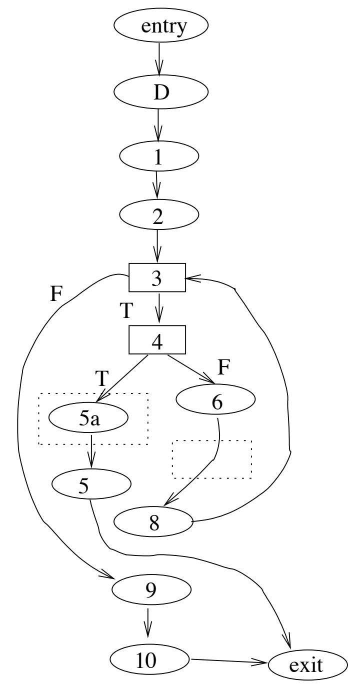
- Answers are shown in the next slides step by step.

OLD AND NEW CFG FOR AVG

```
procedure avg

 1. int count = 0
 2. fread(fileptr,n)
 3. while (not EOF) do
 4.     if (n<0)
 5.         return(error)
        else
 6.         numarray[count] = n
 7.         count++
        endif
 8.     fread(fileptr,n)
    endwhile
 9. avg = calcavg(numarray,count)
10. return(avg)
```




```
procedure avg'

 1. int count = 0
 2. fread(fileptr,n)
 3. while (not EOF) do
 4.     if (n<0)
5a.         print("bad input")
 5.         return(error)
        else
 6.         numarray[count] = n

        endif
 8.     fread(fileptr,n)
    endwhile
 9. avg = calcavg(numarray,count)
10. return(avg)
```



A TEST SUITE FOR PROGRAM AVG

Test Case	Input	Expected Output
1	empty file	0
2	-1	error
3	1 2 3	2

EDGE COVERAGE MATRIX

Edge	Test Case
(entry, 1), (1, 2), (2, 3)	1, 2, 3
(3, 9), (9, 10), (10, exit)	1, 3
(3, 4)	2, 3
(4, 5), (5, exit)	2
(4, 6), (6, 7), (7, 8), (8, 3)	3

- Any tests that exercised edges (4,5) and (6,7) I the old version are selected. T2 and T3 should be rerun for the new version.

PRACTICE QUESTION: RTS #1

```
void fun(int N, int k, int j){
  int sum = 0;
  int product = 1;
  for(i = 1; i <= N; i=i+1){
    sum = sum + i;
    if (k%2==0) product = product*i;
  }
  write(sum);
  write(product);
}
```

```
void fun(int N, int k, int j){
  int sum = 0;
  int product = 1;
  for(i = 1; i <= N; i=i+1){
  sum = sum + i;
  if (k%2==0) product =
product* 2*i;
  }
  write(sum);
  write(product);
}
```

PRACTICE QUESTION: RTS #1

- Draw the control flow graphs for the new version of the program
- Mark the dangerous edges on your control flow graphs.
- Dangerous edges mean the control flow graph edges that have different target nodes in the new version.

PRACTICE QUESTION: RTS #1

- 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.
- Mark if and only if the test must be selected for the new version.
- T1 (N=0, k=1, j=1)
- T2 (N=10, k=2, j=0)
- T3 (N=1, k=3, j=1)
- Answer: T2 should be rerun for the new version.

HARROLD ET AL. RTS FOR JAVA

- Regression Test Selection for Java Software
- OOPSLA 2001
- What are main challenges for making RTS work in Java?
- How did Harrold et al. address challenges for Java software?
- What are differences between this work and Harrold et al.'s RTS for procedural languages?

MAIN CHALLENGES FOR MAKING RTS WORK IN JAVA

- Java language features: in particular, (1) polymorphism, (2) dynamic binding, and (3) exception handling
- Why is polymorphism & dynamic binding difficult to handle in RTS?

MAIN CHALLENGES FOR MAKING RTS WORK IN JAVA

- Java language features: in particular, (1) polymorphism, (2) dynamic binding, and (3) exception handling
- Why is polymorphism & dynamic binding difficult to handle in RTS?
- The target of method calls depends on the dynamic type of a receiver object.

- Java language features: in particular, (1) polymorphism, (2) dynamic binding, and (3) exception handling
- Why is polymorphism & dynamic binding difficult to handle in RTS?
  - The target of method calls depends on the dynamic type of a receiver object.

```
class B extends A {
};
class C extends B {
 public void m(){...};
};
void bar(A p) {
  A.foo();
  p.m();
}
```

```
class B extends A {
 public void m(){...};
};
class C extends B {
 public void m(){...};
};
void bar(A p) {
  A.foo();
  p.m();
}
```

EXTERNAL LIBRARIES AND COMPONENTS

- Why is it important to model interaction between the main code and its libraries?
- External library code can invoke internal methods if the internal methods override external methods.

- External libraries and components
  - Why is it important to model interaction between the main code and its libraries?
  - External library code can invoke internal methods if the internal methods override external methods.

```
class B extends A {
  public void foo() {...};
}
class C extends B {
 public void bar() {...};
};
```

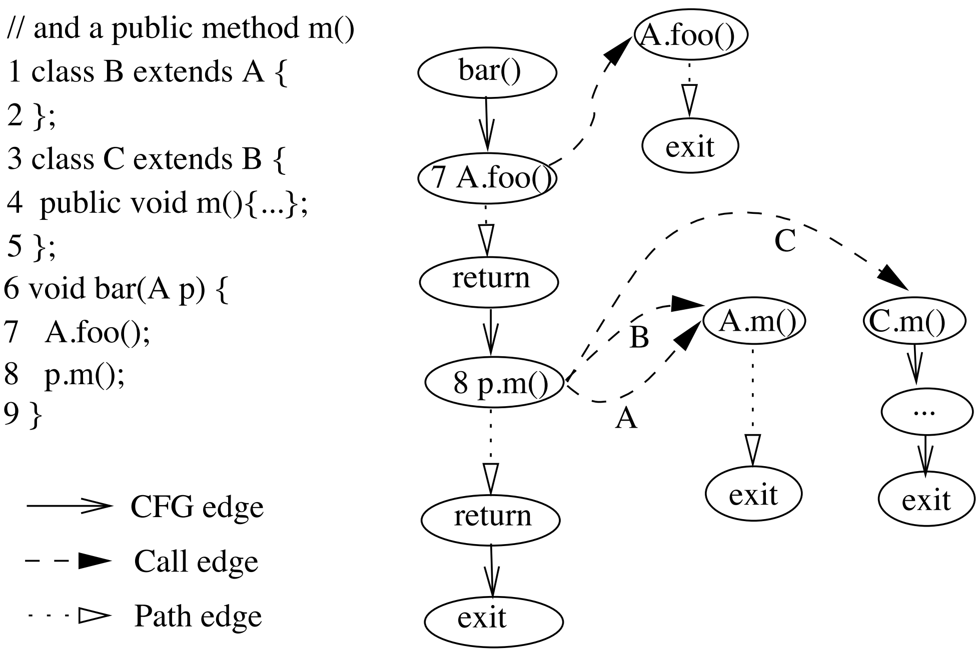
```
class B extends A {
  public void foo() {...};
  public void bar() {...};
}
class C extends B {

};
```

JIG (JAVA INTERCLASS GRAPH)

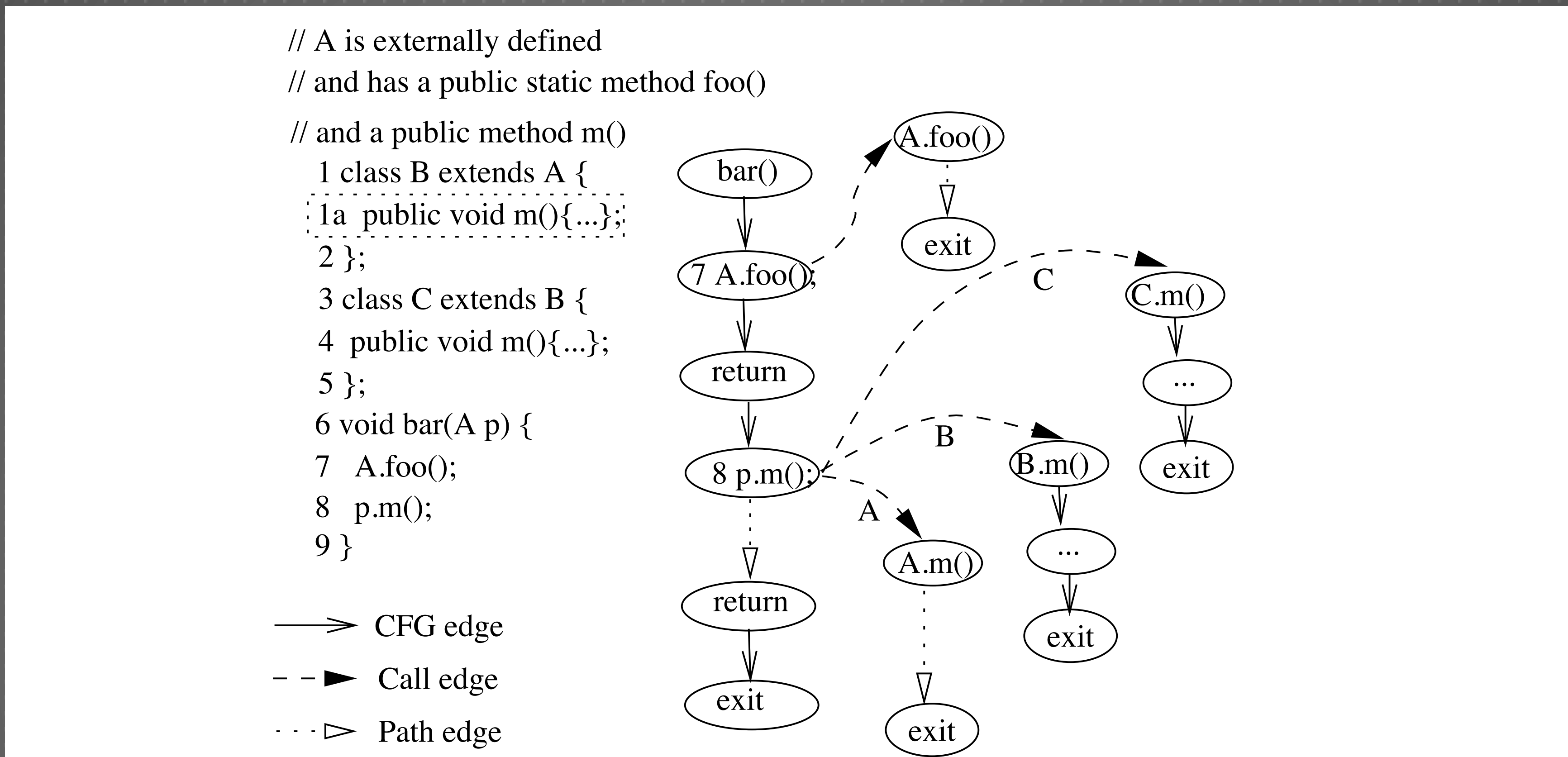
- JIG extends CFG to handle five kinds of Java features
- (1) variable and object type information
- (2) internal and external methods
- (3) interprocedural interactions through calls to internal methods or external methods from internal methods.
- (4) interprocedural interactions through calls to internal methods from external methods
- (5) exception handling

OLD CFG WITH DYNAMIC DISPATCHING

```
// A is externally defined
// and has a public static method foo()
// and a public method m()
1 class B extends A {
2 };
3 class C extends B {
4  public void m(){...};
5 };
6 void bar(A p) {
7   A.foo();
8   p.m();
9 }
```



NEW CFG FOR DYNAMIC DISPATCHING



DANGEROUS EDGES

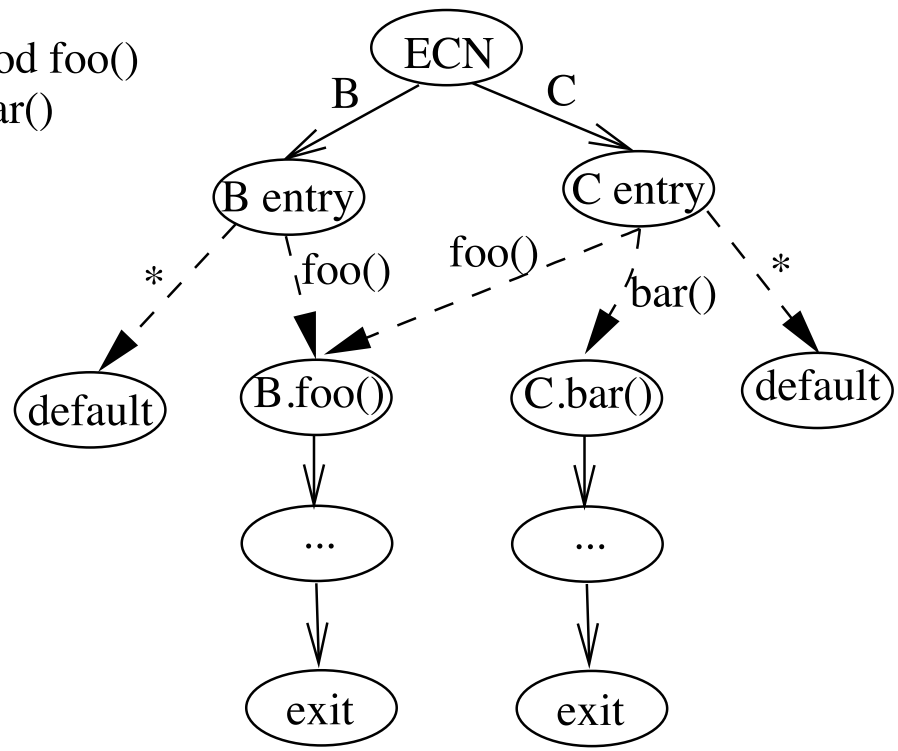
- Calling p.m() on an object with type B

OLD INTERACTION GRAPH

```
//A is externally defined
// and has a  public method foo()
// and a public method bar()

class B extends A {
  public void foo() {...};
}
class C extends B {
 public void bar() {...};
};
```

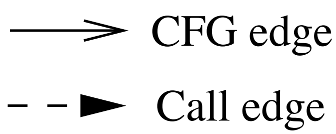
CFG edge

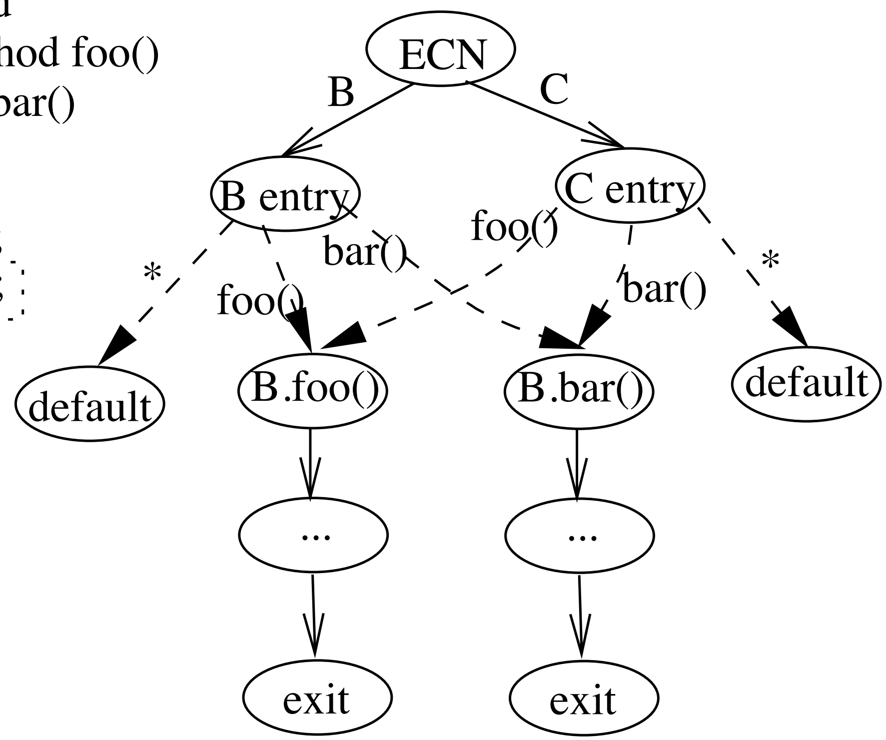
Call edge



NEW INTERACTION GRAPH

```
//A is externally defined
// and has a  public method foo()
// and a public method bar()

class B extends A {
  public void foo() {...};
  public void bar() {...};
}
class C extends B {

};
```





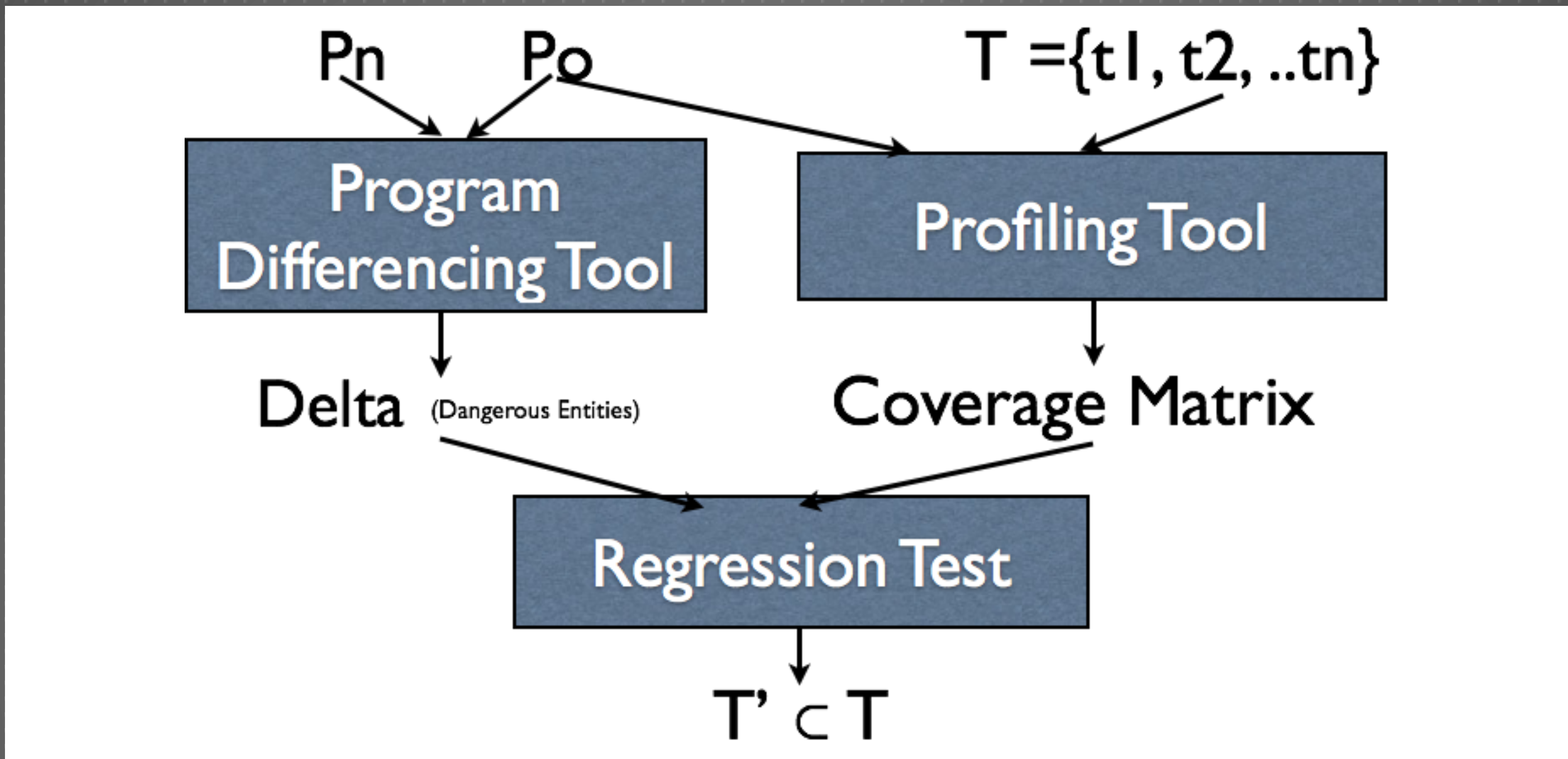
DANGEROUS EDGES

- Calling bar() on object of type B
- Calling bar() on object of type C

JAVA RTS ALGORITHM

- start from either main () node, the ECN node, static method entries,
- the algorithm traverses the JIgs and add the dangerous edges that it finds to E.
- It marks N as "N visited" to avoid comparing N and N' again in a subsequent iteration
- If the target along the same edge is different between two graphs, then it becomes a dangerous edge.
- One way to determine the equivalence of two nodes is to examine the lexicographic equivalence of the text associated with the two nodes.

RECAP: RTS FRAMEWORK



RECAP

- We have studied sub-problems within regression testing.
- We have studied an algorithm for regression test selection.

QUESTIONS?