
COMP5111 – Fundamentals of Software Testing and Analysis

Object-Oriented & Regression Testing



Shing-Chi Cheung

Computer Science & Engineering

HKUST

Declared Type vs Actual Type

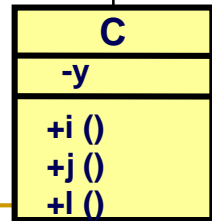
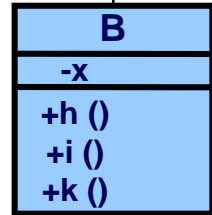
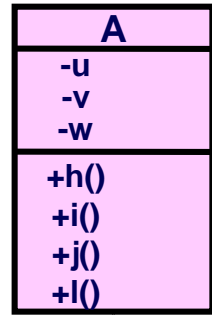
Declared type: The type given when an object reference is declared

`Clock w1; // declared type Clock`

Actual type: The type of the current object
`w1 = new Watch(); // actual type Watch`

Example DU Pairs and Anomalies

Consider what happens when an overriding method has a different def-set from that of its overridden method



Method	Defs	Uses
A::h()	{A::u, A::w}	
A::i()		{A::u}
A::j()	{A::v}	{A::w}
A::l()		{A::v}
B::h()	{B::x}	
B::i()		{B::x}
C::i()		
C::j()	{C::y}	{C::y}
C::l()		{A::v}

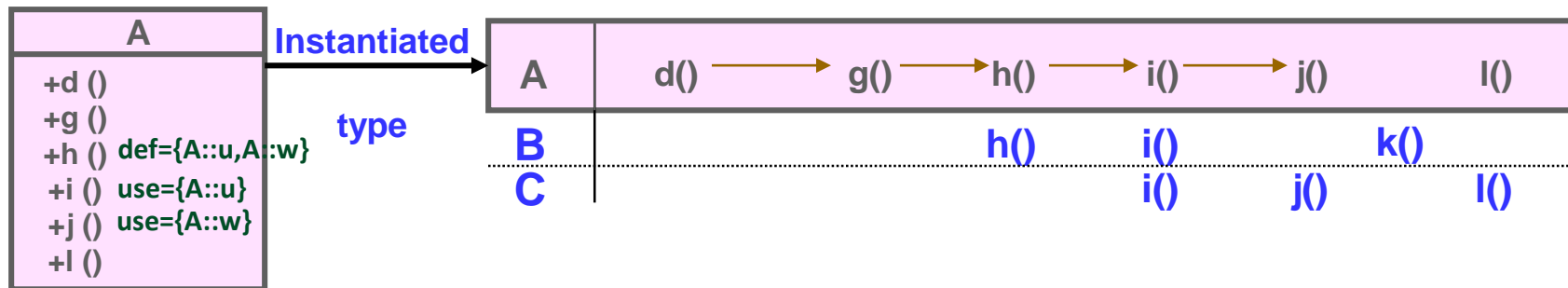
DU anomaly

```
void f(A o) {
    o.h(); o.i();
    o.j(); o.l();
}
```

DU anomaly

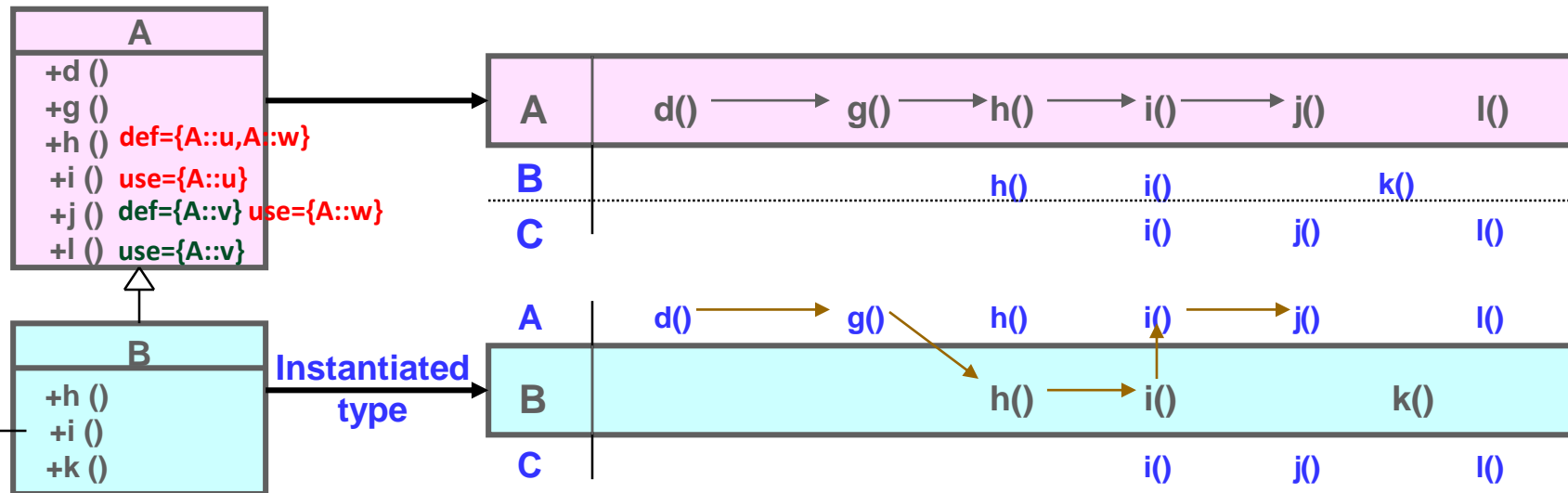
A::h() assumes calling j() later; B::h() does not assume so

Polymorphism Visualization (Yo-Yo Graph)



Object is of type A
A::d ()

Polymorphism Headaches (Yo-Yo)



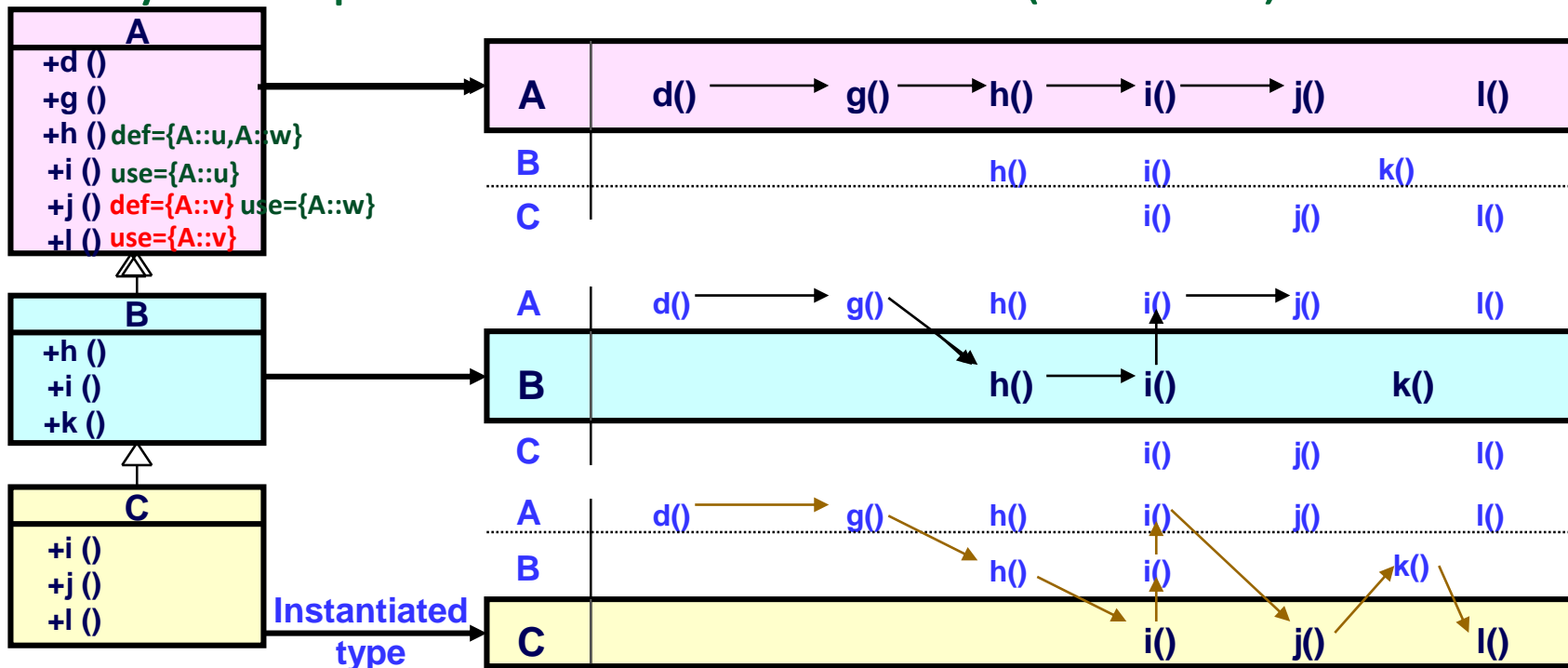
`void i() { super.i(); ... }`

Object is of type B
B::d()

Data flow anomaly of A::u occurs at the call of A::i()
Data flow anomaly of A::w occurs at the call of A::j()

Note: Testing the methods in B alone is inadequate. We need to consider the methods in A as well.

Polymorphism Headaches (Yo-Yo)

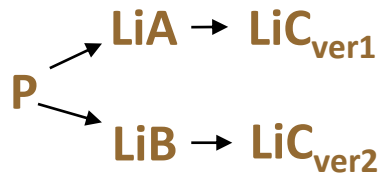


Data flow anomaly of A::v occurs at the call of C::l()

Common OO Faults

Causes of Faults in OO Programs

- Complexity is relocated to the connections among components
- Inheritance and Polymorphism yield vertical and dynamic integration
 - Deep and multiple inheritance chains on large libraries with dependency conflicts
- Many faults can now only be detected at runtime
- Aggregation and use relationships are more complex
- Designers do not carefully consider visibility of data and methods

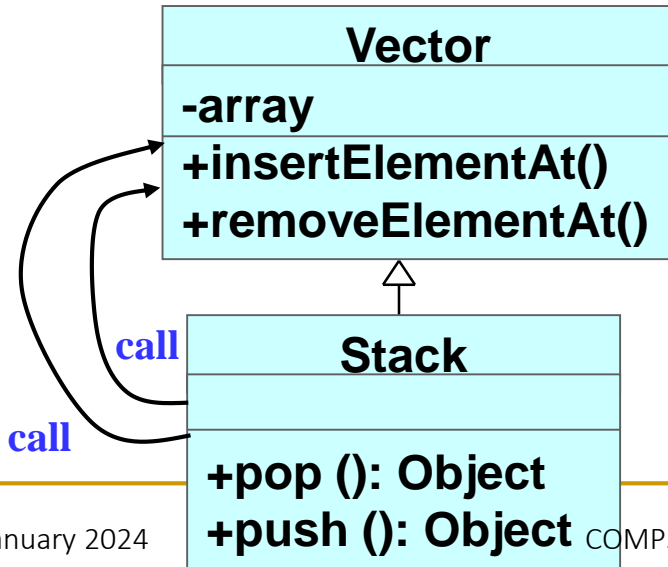


Object-oriented Faults

- Only consider **faults** that arise as a direct result of OO language features:
 - inheritance
 - polymorphism
 - constructors
 - visibility
- **Language independent** (as far as possible)

Inconsistent Type Use (ITU)

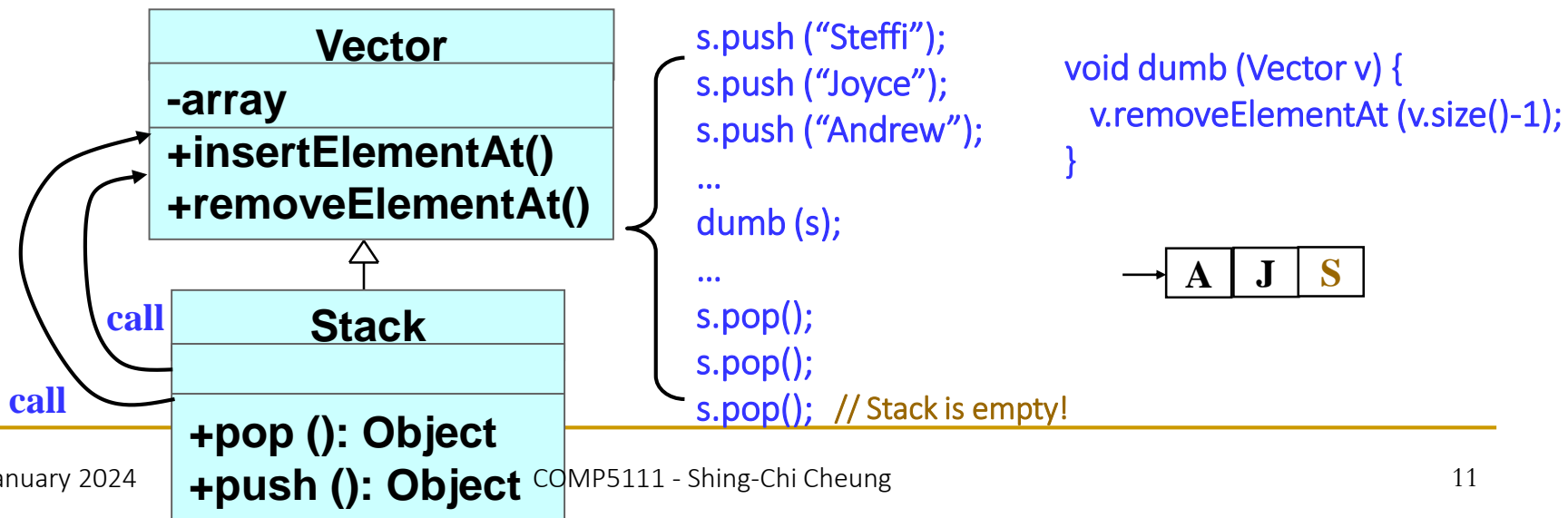
- No **overriding** methods (no polymorphism)
- C extends T , and C adds new methods (**extension**)
- An **object is used** “as a C ”, then as a T , then as a C
- Methods in T can put object in state that is **inconsistent** for C



```
class Stack extends Vector {
    public int count = 0;
    public Object pop() {
        --count;
        return super.removeElementAt();
    }
    public Object push() {
        ++count;
        return super.insertElementAt();
    }
}
```

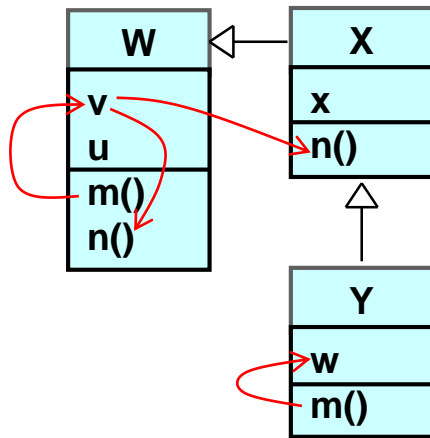
Inconsistent Type Use (ITU)

- No **overriding** methods (no polymorphism)
- C extends T , and C adds new methods (**extension**)
- An **object is used** “as a C ”, then as a T , then as a C
- Methods in T can put object in state that is **inconsistent** for C



State Definition Anomaly (SDA)

- Y extends W , and Y overrides some methods
- The overriding methods in Y fail to define some variables that the overridden methods in W defined



- $W::m()$ defines v and $W::n()$ uses v
- $X::n()$ uses v
- $Y::m()$ does not define v

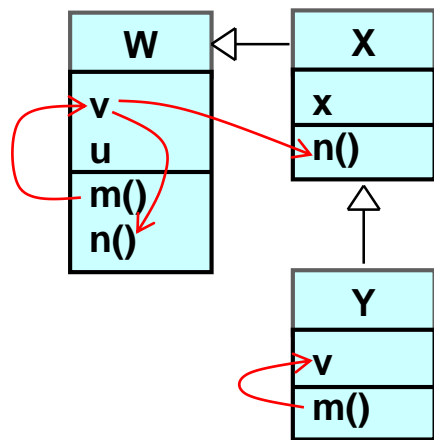
```
void f(W o) {  
    o.m();  
    o.n();  
}
```

For an object of type Y , a data flow anomaly of v exists and results in a fault if $m()$ is called, then $n()$

Note that it works for objects of type X . This suggests that testing $Y::m()$ alone in regression testing is inadequate.

State Definition Inconsistency (SDIH)

- Hiding a variable, possibly accidentally
- If the descendant's version of the variable is defined, the ancestor's version may not be



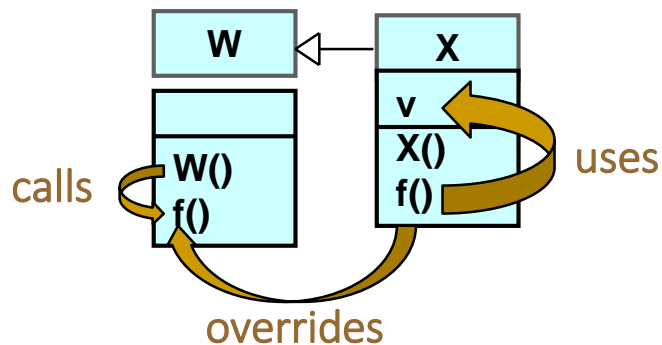
- Y hides W 's version of v
- $Y::m()$ defines $Y::v$
- $X::n()$ uses v

```
void f(W o) {  
    o.m();  
    o.n();  
}
```

For an object of type Y , a data flow inconsistency of v may exist and result in a fault if $m()$ is called, then $n()$

Anomalous Construction Behavior (ACB1)

- Constructor of W calls a method $f()$
- A child of W , X , overrides $f()$
- $X::f()$ uses variables that should be defined by X 's constructor

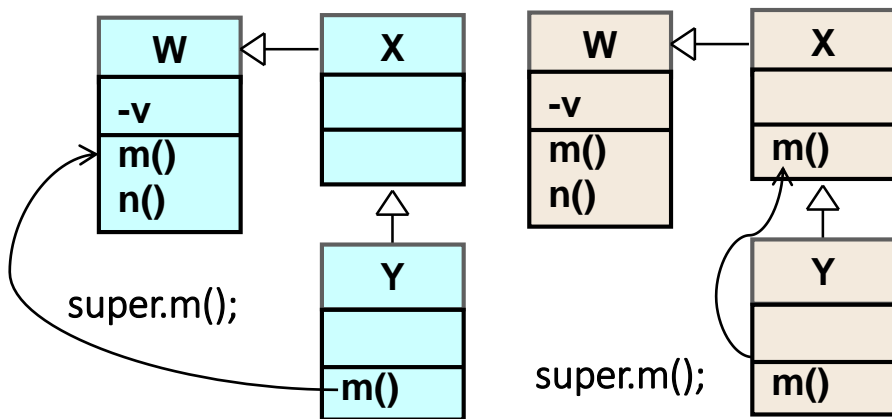


When an object of type X is constructed, $W()$ is run before $X()$.

When $W()$ calls $X::f()$, v is used, but it has not yet been given a value!

State Visibility Anomaly (SVA)

- A private variable v is **declared** in ancestor W , and v is defined by $W::m()$
- X **extends** W and Y extends X
- Y overrides $m()$, and **calls** $W::m()$ to define v

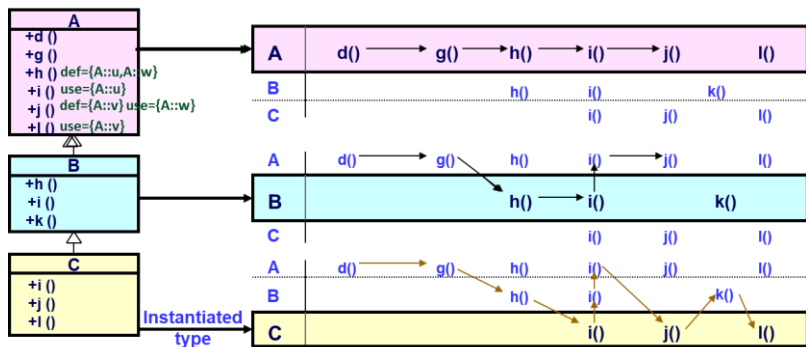


$X::m()$ is added later

$Y::m()$ can no longer call $W::m()$ directly, and has no direct control over the dataflow anomaly of v .

Make such edge pair a test requirement.

Summary



Do we need additional test requirements for OO programs?

- We build an OO call graph denoting possible method calling relations between the methods in super- and sub-classes
- A yoyo graph is a path of the OO call graph
- We then apply various graph-based test requirements

Grove, David & DeFouw, Greg & Dean, Jeffrey & Chambers, Craig. (1997). Call Graph Construction in Object-Oriented Languages. *ACM Transactions on Programming Languages and Systems - TOPLAS*. 32. 108-124. 10.1145/263700.264352.

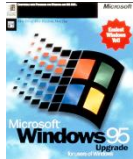
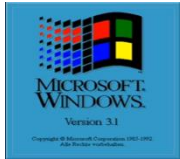


Regression Testing

<https://youtu.be/aeu5zacsHsl> (1:09 min)

What is regression testing?

- Successful projects keep evolving their code.
- Code must be re-validated after change.
- This process is expensive and occurs repeatedly.
- Regression testing studies how to re-validate a program in a cost-effective way after change.



Regression testing



1. Develop P
2. Test P
3. Release P
4. Modify P to P'
5. Test P' for new functionality
6. Perform regression testing on P' to ensure that the code carried over from P behaves correctly
7. Release P'

What tests to use?

Option 1:

All valid tests from the previous version and new tests created to test any added functionality. [This is the TEST-ALL approach.]

What are the strengths and shortcomings of this approach?

TEST-ALL:

- + Safe
- Resource consuming
- Lots of redundant tests
- It can take days or weeks

Industry Adoption of Continuous Integration

	Google DS	Rails DS
# of total commits	4,421	2,804
# of failing commits	1,022	574
Avg commit duration (sec)	948	1,505
# of distinct test suites	5,536	2,072
# of distinct failing test suites	154	203
Avg # of test suites per commit	331	1,280
# of total test suite executions	1,461,303	3,588,324
# of failing test suite executions	4,926	2,259
Test suite execution time (sec)	4,192,794	4,220,482
	1,164 cpu-hrs	1,172 cpu-hrs

- Google DS contains info based on a sample of Google products over 15 days
- Rails DS contains info based on a popular open source project Rails (from Mar-Aug/2016)
- Commit duration = Time taken to run test suites for a commit
- Time to validate a fix can be non-trivial
- Even more expensive if including the building time

Source: Liang et al., Redefining Prioritization: Continuous Prioritization for Continuous Integration, ICSE18.

Test selection

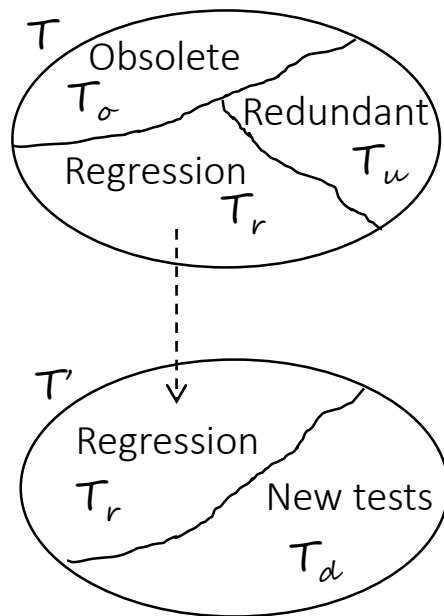
Option 2:

Select a regression test set \mathcal{T}_r of the original test set \mathcal{T} such that successful execution of the modified code \mathcal{P}' against \mathcal{T}_r implies that all the functionality carried over from the original code \mathcal{P} to \mathcal{P}' is intact.

$$\mathcal{T}_r \subseteq \mathcal{T} \text{ such that } \mathcal{T}_r(\mathcal{P}') \Rightarrow \mathcal{T}(\mathcal{P})$$

\mathcal{T}_r can be found using several methods. We will discuss two of these known as **test minimization** and **test prioritization**.

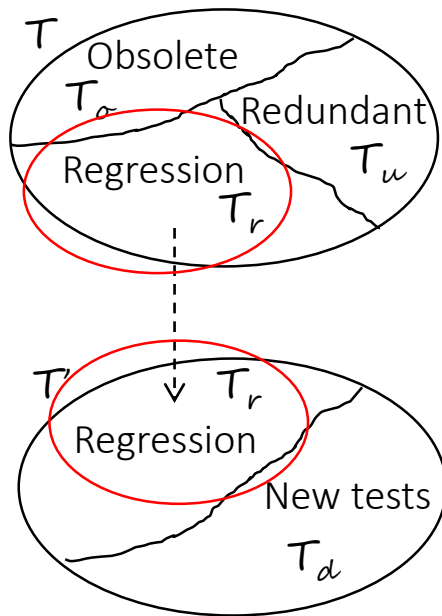
Regression Test Selection problem



Given test set T , our goal is to determine T_r such that successful execution of P' against T_r implies that modified or newly added code in P' has not broken the code carried over from P .

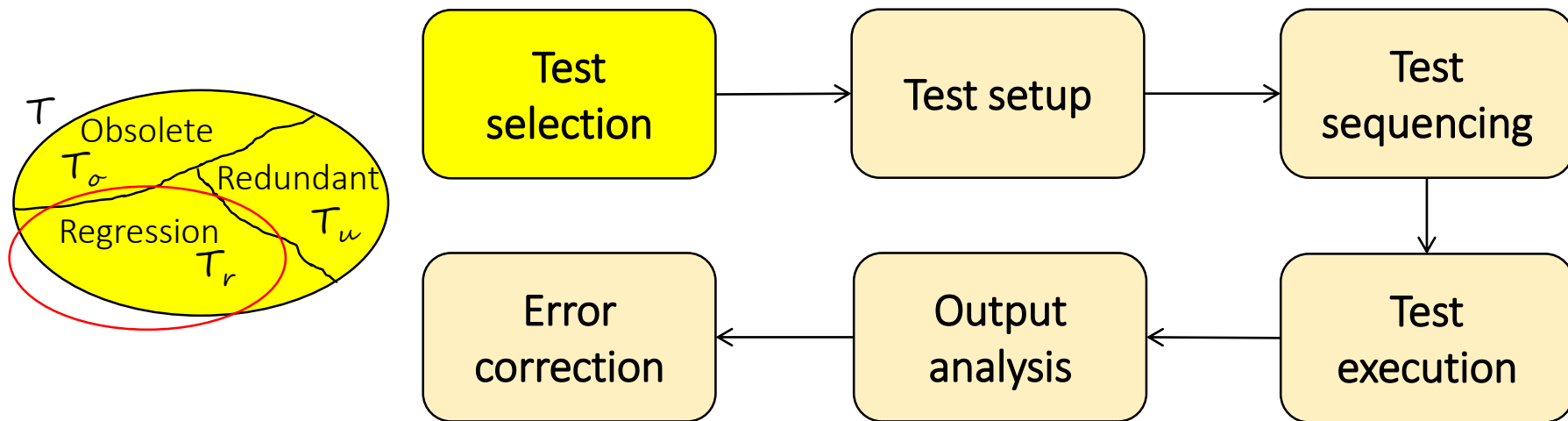
Note that some tests might become obsolete when P is modified to P' . Such tests are not included in the regression subset T_r . The task of identifying such obsolete tests is known as **test revalidation**.

Regression Test Selection problem



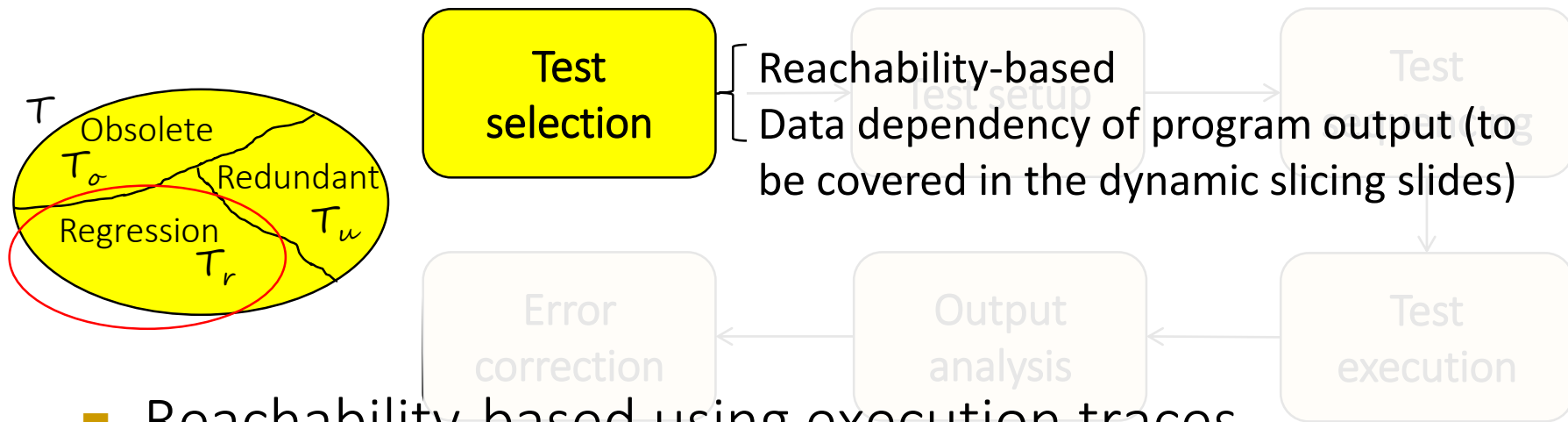
- Obsolete tests
 - Functionality to be tested are obsolete
- Redundant tests
 - Their test coverage is subsumed by either regression tests or new tests in the revised version
- Regression tests
 - Tests for functionality to be preserved
- New tests
 - Tests created for new functionality

Regression Test Process



- Most research studies focus on test selection

Regression Test Process



- Reachability-based using execution traces
- Data dependency of program output using dynamic slices

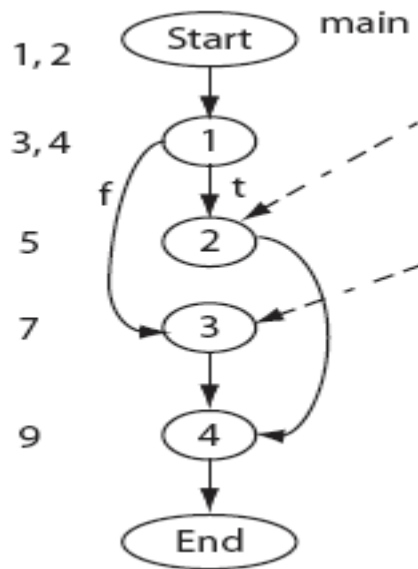
Reachability-based test selection

Test Selection – Example Program

```
1  main(){          1  int g1(int a, b){  1  int g2 (int a, b){
2  int x,y,p;        2  int a,b;          2  int a,b;
3  input (x,y);      3  if(a+ 1==b)        3  if(a==(b+1))
4  if (x<y)          4    return(a*a);    4    return(b*b);
5    p=g1(x,y);      5  else                5  else
6  else              6    return(b*b);    6    return(a*a);
7    p=g2(x,y);      7  }                7  }
8  endif
9  output (p);
10 end
11 }
```

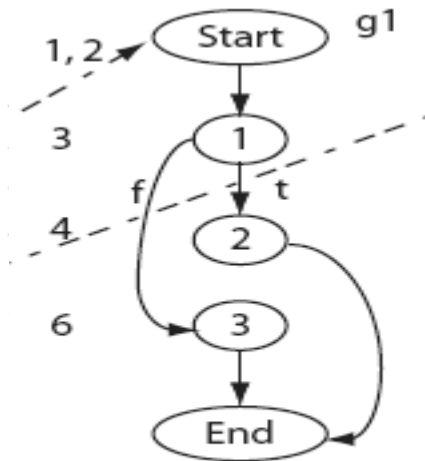
Test Selection (1) – CFG Construction

```
1  main(){  
2  int x,y,p;  
3  input (x,y);  
4  if (x<y)  
5    p=g1(x,y);  
6  else  
7    p=g2(x,y);  
8  endif  
9  output (p);  
10 end  
11 }
```



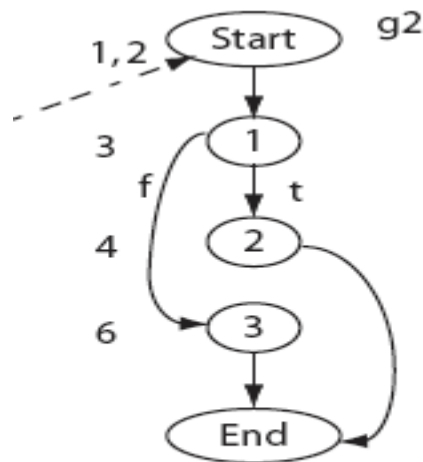
Test Selection (1) – CFG Construction

```
1  int g1(int a, b){  
2  int a,b;  
3  if(a+ 1==b)  
4    return(a*a);  
5  else  
6    return(b*b);  
7  }
```



Test Selection (1) – CFG Construction

```
1  int g2 (int a, b){  
2  int a,b;  
3  if(a==(b+1))  
4    return(b*b);  
5  else  
6    return(a*a);  
7  }
```



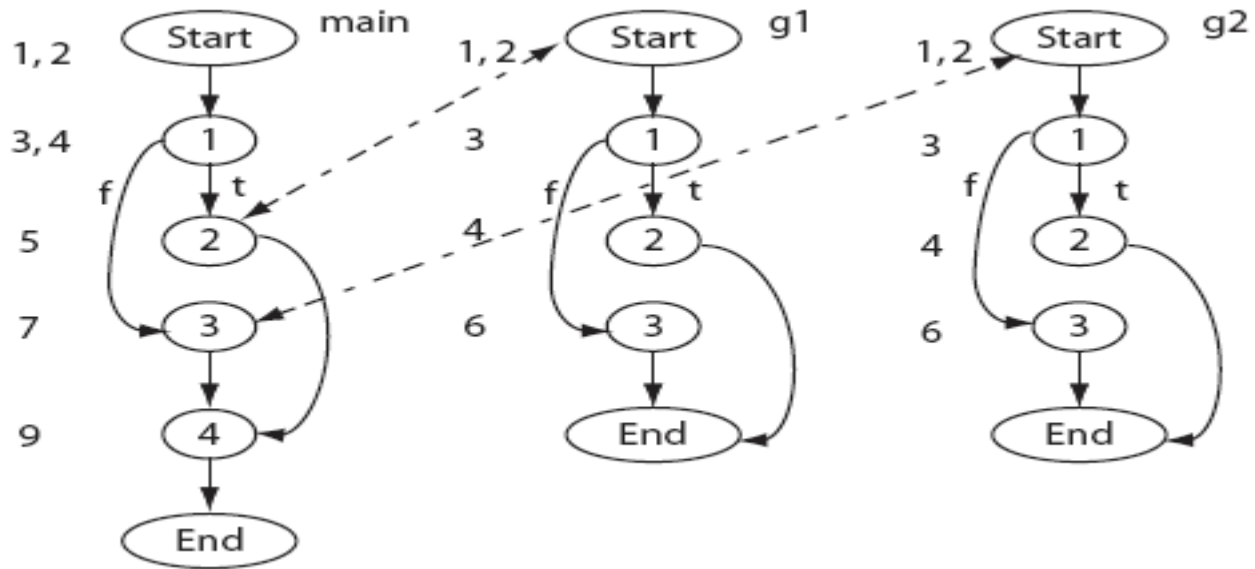
Test Selection (2) – Obtain Execution Traces

An **execution trace** of program P for some test t in T is the

sequence of nodes in G traversed when P is executed against t

Suppose $T = \{t_1, t_2, t_3\}$ is the test suite that covers the required test requirements, where

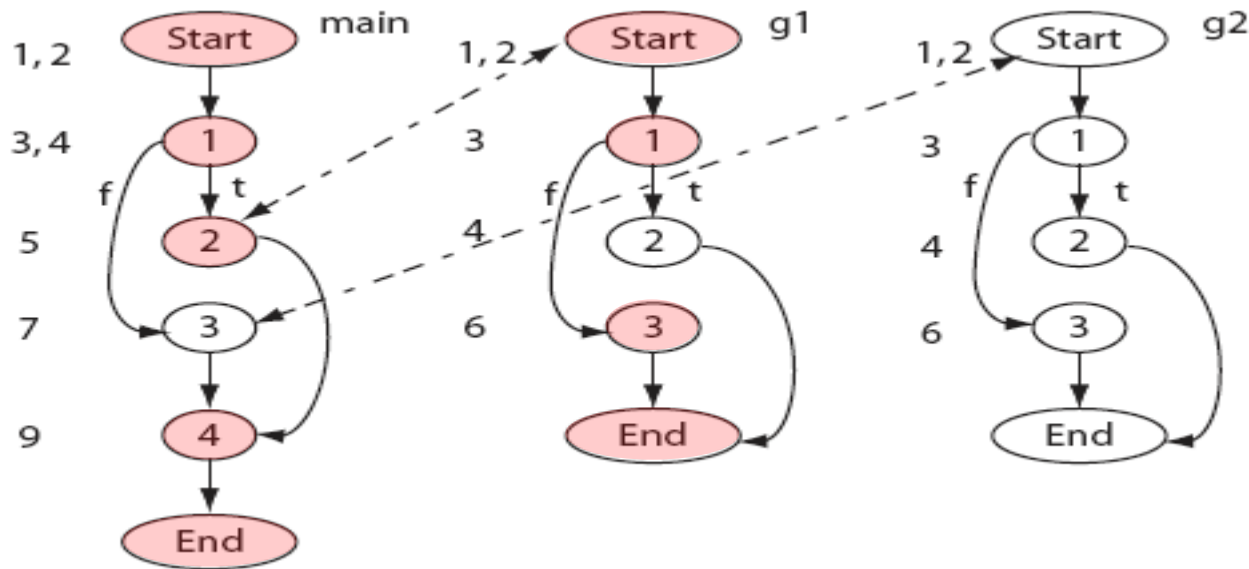
$$T = \left\{ \begin{array}{l} t_1 : \langle x = 1, y = 3 \rangle \\ t_2 : \langle x = 2, y = 1 \rangle \\ t_3 : \langle x = 3, y = 1 \rangle \end{array} \right\}$$



Test Selection (2) – Obtain Execution Traces

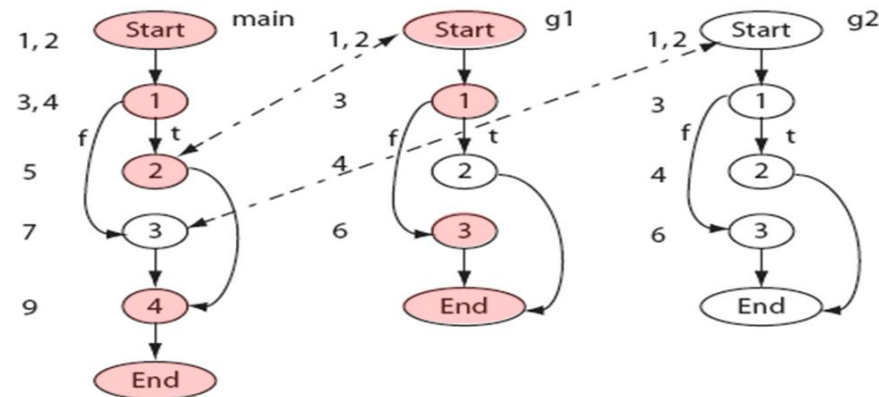
t_1 : main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End

$$T = \left\{ \begin{array}{l} t_1 : \langle x = 1, y = 3 \rangle \\ t_2 : \langle x = 2, y = 1 \rangle \\ t_3 : \langle x = 3, y = 1 \rangle \end{array} \right\}$$



Test Selection (2) – Obtain Execution Traces

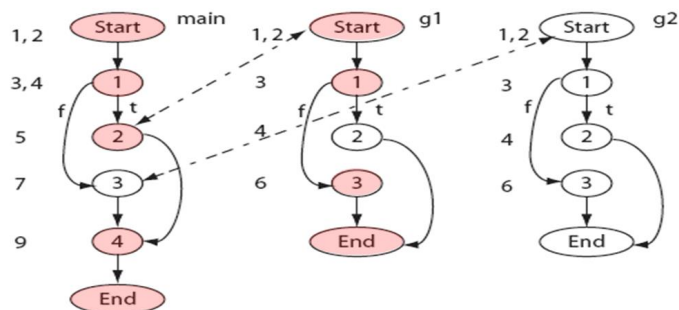
$$T = \left\{ \begin{array}{l} t_1 : \langle x = 1, y = 3 \rangle \\ t_2 : \langle x = 2, y = 1 \rangle \\ t_3 : \langle x = 3, y = 1 \rangle \end{array} \right\}$$



Test (t)	Execution trace ($trace(t)$)
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

Test Selection (3) – Extract Test Vectors

A test vector for node n , denoted by $\text{test}(n)$, is the set of tests that traverse node n in the CFG. For program P , we obtain the following test vectors.

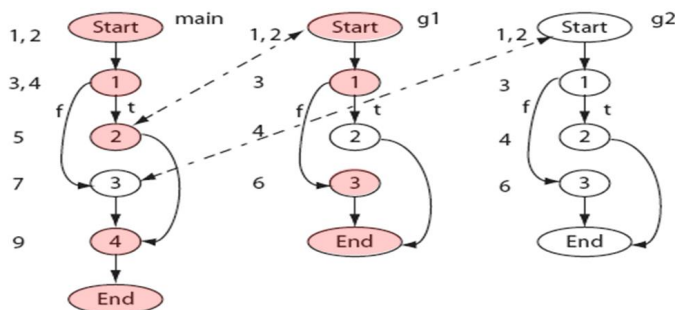


Function	Test vector ($\text{test}(n)$) for node n			
	1	2	3	4
main	t_1, t_2, t_3	t_1, t_3	t_2	t_1, t_2, t_3
g1	t_1, t_3	t_3	t_1	—
g2	t_2	t_2	None	—

Test Selection (3) – Extract Test Vectors

Test (t)	Execution trace ($trace(t)$)
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

t_1, t_2, t_3 can reach
node 4 in main



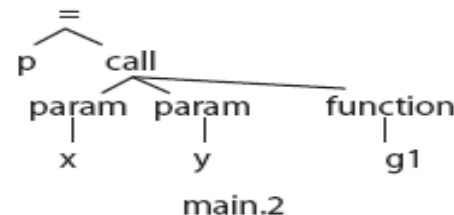
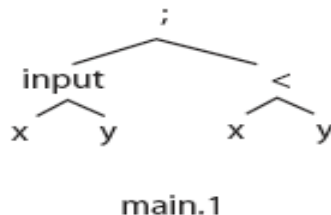
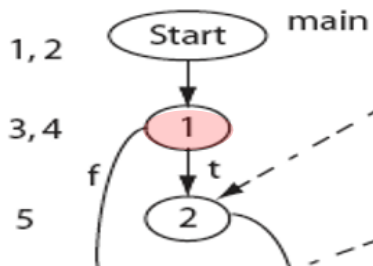
Function	Test vector ($test(n)$) for node n			
	1	2	3	4
main	t_1, t_2, t_3	t_1, t_3	t_2	t_1, t_2, t_3
g1	t_1, t_3	t_3	t_1	—
g2	t_2	t_2	None	—

Test Selection (4) – Construct Syntax trees

A syntax tree is constructed for each node of CFG(P) and CFG(P').

Recall that each node represents a basic block. Here sample syntax trees for the example program.

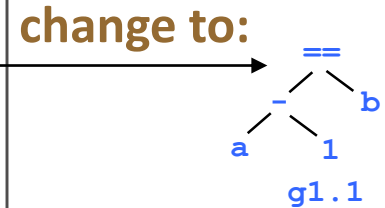
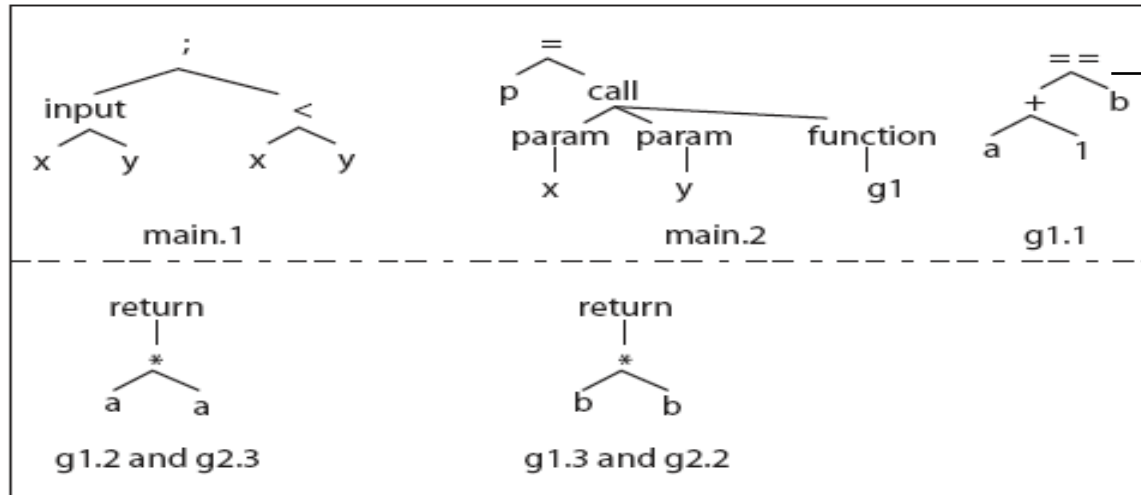
```
1  main(){
2  int x,y,p;
3  input (x,y);
4  if (x<y)
5    p=g1(x,y);
6  else
```



Test Selection (4) – Construct Syntax trees

A syntax tree is constructed for each node of CFG(P) and CFG(P').

Syntax trees for CFG(P):



Test selection (5) – Determine T' for P'

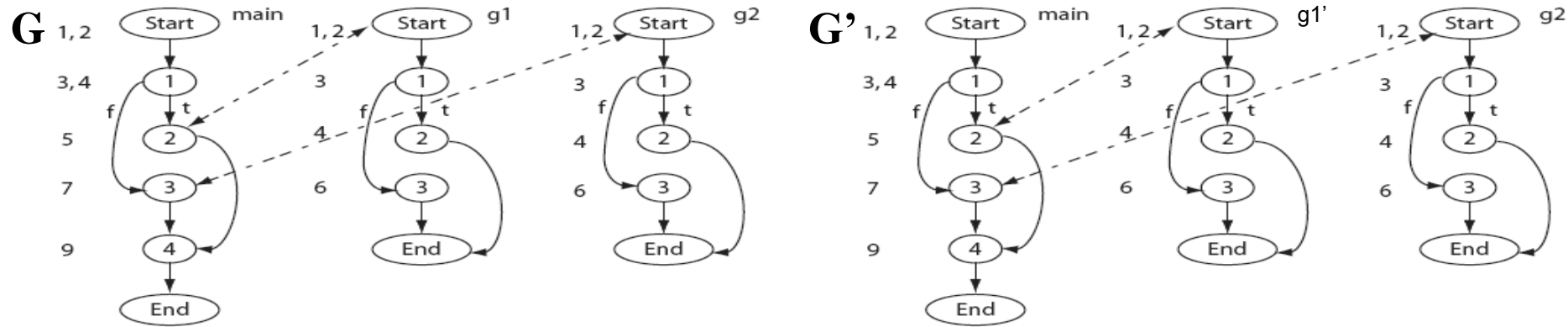
Given the execution traces and the CFGs for P and P' , the following three steps are executed to obtain a subset T' of T for regression testing of P' .

Step 1 Set $T' = \Phi$. Unmark all nodes in G and in its child CFGs.

Step 2 Call procedure **SelectTests**($G.\text{Start}$, $G'.\text{Start}'$), where $G.\text{Start}$ and $G'.\text{Start}'$ are, respectively, the start nodes in G and G' .

Step 3 T' is the desired test set for regression testing P' .

Test selection (5) – Step 1

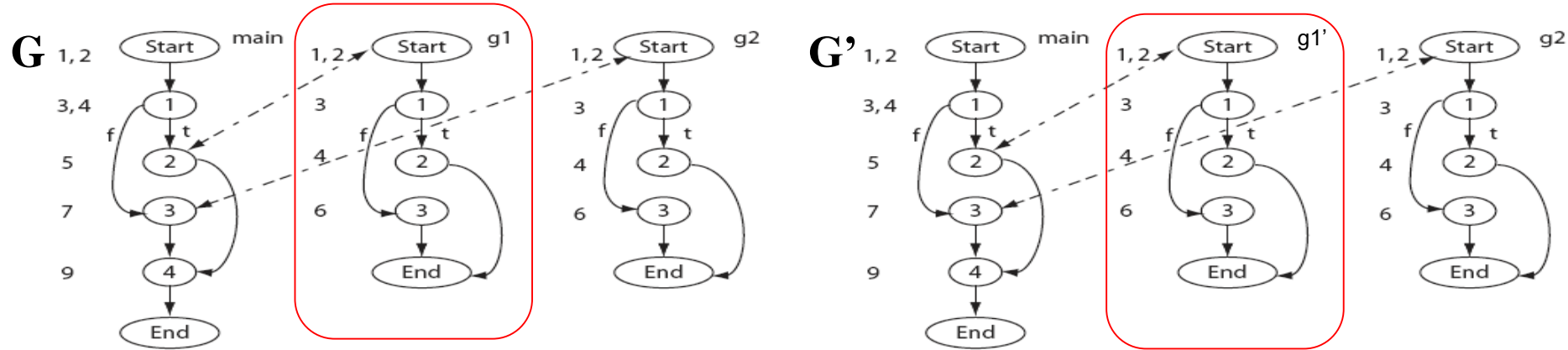


Step 1 Set $T' = \Phi$. Unmark all nodes in G and in its child CFGs.

Step 2 Call procedure **SelectTests**(G .Start, G' .Start'), where G .Start and G' .Start' are, respectively, the start nodes in G and G' .

Step 3 T' is the desired test set for regression testing P' .

Test selection (5) – Step 2: SelectTests

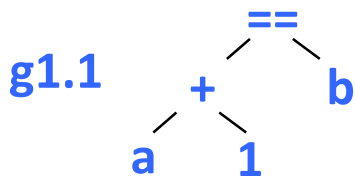


The procedure proceeds in parallel and the corresponding nodes are compared. If two nodes N in $CFG(P)$ and N' in $CFG(P')$ are found to be syntactically different, all tests in $test(N)$ are added to T' .

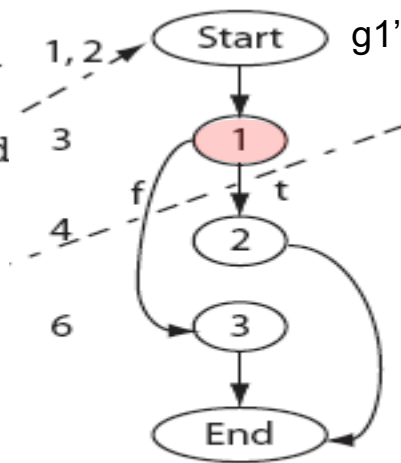
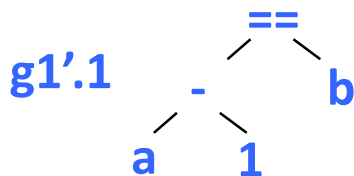
Test selection (5) – Step 2: SelectTests

Suppose that function g1 in P is modified as follows.

```
1 int g1(int a, b){  
2 int a,b;  
3 if(a+ 1==b)  
4   return(a*a);  
5 else  
6   return(b*b);  
7 }
```



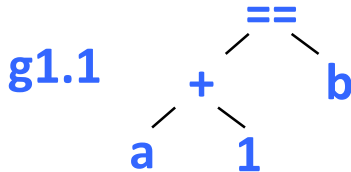
```
1 int g1(int a, b){ ← Modified g1  
2 int a, b;  
3 if(a-1==b) ← Predicate modified  
4   return(a*a),  
5 else  
6   return(b*b),  
7 }
```



Test selection (5) – Step 2: SelectTests

Suppose that function g1 in P is modified as follows.

1 int g1(int a, b){
2 int a,b;
3 if(a+ 1==b)
4 return(a*a);
5 else
6 return(b*b);
7 }

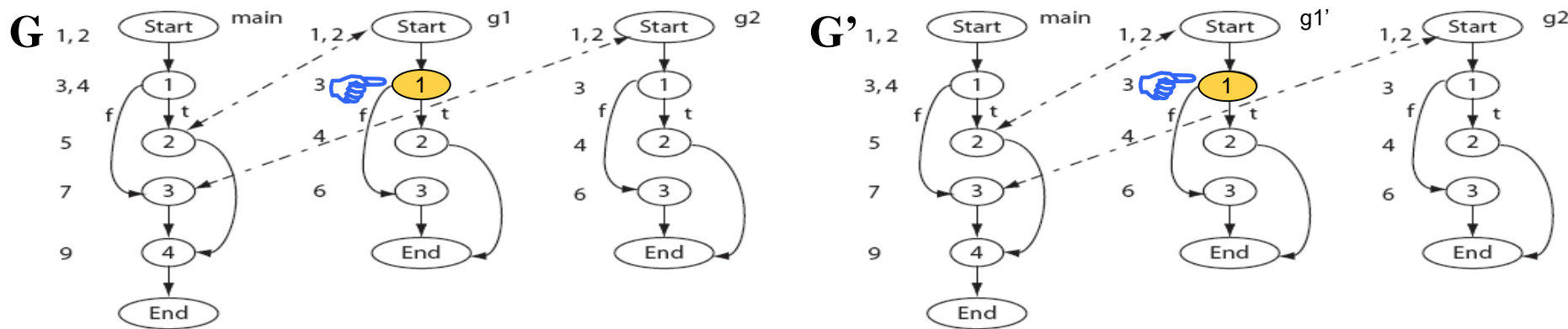


1 int g1(int a, b){ ← Modified g1.
2 int a, b;
3 if(a-1==b) ← Predicate modified.
4 return(a*a),
5 else



Function	Test vector (<i>test(n)</i>) for node <i>n</i>			
	1	2	3	4
main	t_1, t_2, t_3	t_1, t_3	t_2	t_1, t_2, t_3
g1	t_1, t_3	t_3	t_1	—
g2	t_2	t_2	None	—

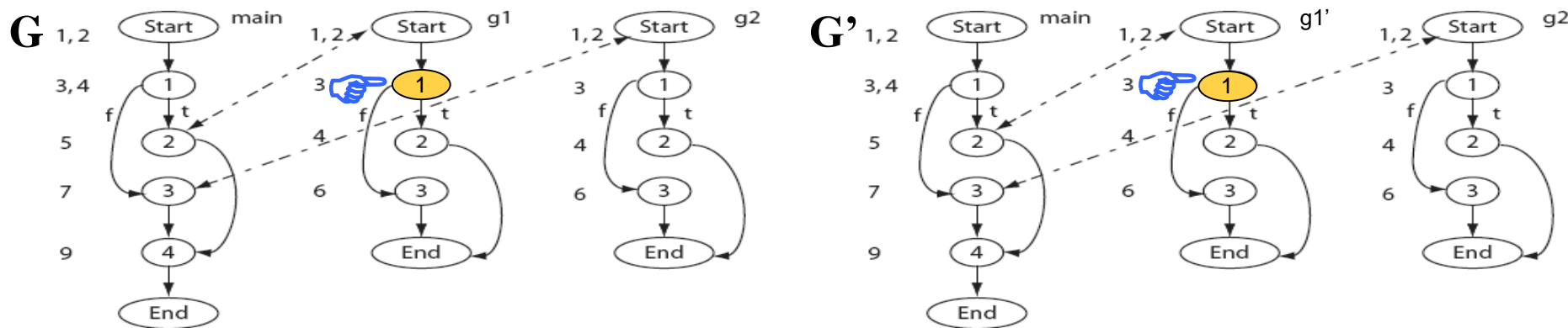
Test selection (5) – Step 2: SelectTests



SelectTests algorithm updates T' from $\{\}$ to $\{t1, t3\}$

SelectTests algorithm continues until it has compared all node pairs

Test selection (5) – Step 3



Output regression test suite $T' = \{t1, t3\}$ after comparing all node pairs

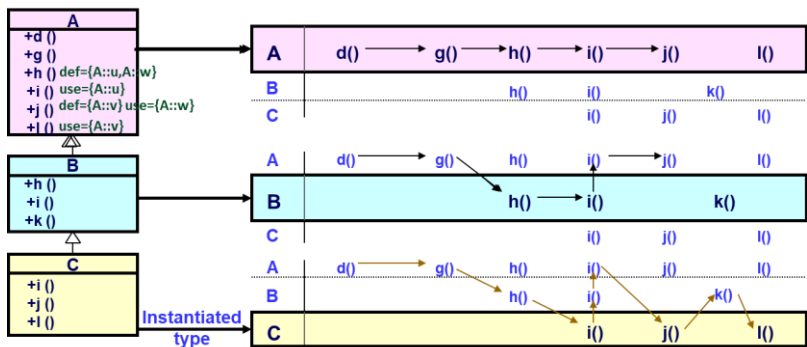
Note: All tests covering the tagged node(s) are selected

Test Suite Reduction using Coverage

Coverage-based Test Reduction

- A regression test suite may be further reduced based on the coverage of changed test requirements
- Changed test requirements can be program statements, branches, predicates, def-use chains, functions and mutants
- Illustration: R1, R2 and R3 are test requirements
 - Test t1: covers R1 and R2
 - Test t2: covers R2 and R3
 - Test t3: covers R1, R2 and R3
 - Suppose code change affects R1 and R3
 - Test suite reduction by coverage: $\{t1, t2, t3\} \rightarrow \{t3\}$

Regression testing of OO Programs



Couldn't we simply test those methods got changed?

- Method changes in an OO program can induce changes in call relations (e.g., super.f()) and alter the call graph
- Identify changed test requirements (e.g., edges, du-paths)
- Tests are selected if they cover the changed test requirements



Tools

- A field not adequately covered by open-source tools.
- Selenium (free) – for web applications
- Testsigma (commercial)
- Katalon (commercial)
- Ranorex Studio (commercial)
- Developers tend to re-run all JUnit tests after code modification.
 - Some refer to this as “smoke” tests.
 - Some re-run only those JUnit tests for changed classes.
 - Inherit JUnit tests from superclasses to subclasses.

Test and oracle generation for regression testing

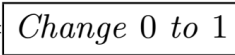
Can we generate effective tests and oracles to detect regression faults?



Enhanced test oracle generation for regression testing

Gordon Fraser and Andreas Zeller. Mutation-driven Generation of Unit Tests and Oracles, ISSTA 2010.

Revisit search-based test generation

```
1 public class LocalDate {  
2   // The local milliseconds from 1970-01-01T00:00:00  
3   private long iLocalMillis;  
4   ...  
5   // Construct a LocalDate instance  
6   public LocalDate(Object instant, Chronology c) {  
7     ... // convert instant to array values  
8     ... // get iChronology based on c and instant  
9     iLocalMillis = iChronology.getDateTimeMillis(  
10      values[0], values[1], values[2], 0);   
11   }  
12 }
```

An initialization module in JodaTime

The subtle behavioral change cannot be detected by the tests bundled with JodaTime

```
1 LocalDate var0 = new org.joda.time.LocalDate()  
2 DateTime var1 = var0.toDateTimeAtCurrentTime()  
3 LocalDate var2 = new org.joda.time.LocalDate(var1)  
4 assertTrue(var2.equals(var0));
```

Generated test that detects the change

Is it possible to generate tests (including assertions) to detect tiny code changes?

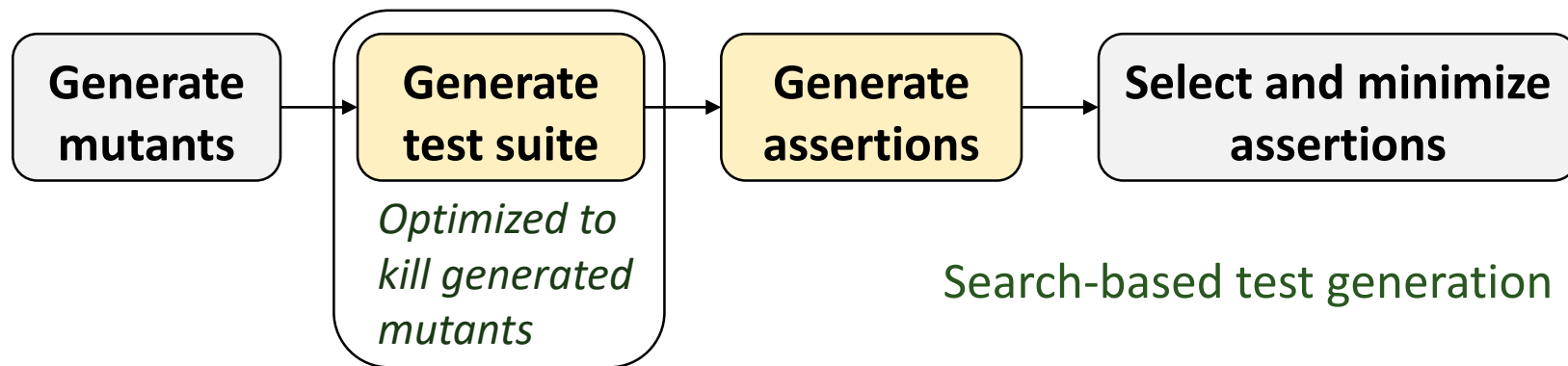
Gordon Fraser and Andreas Zeller. Mutation-driven Generation of Unit Tests and Oracles, ISSTA 2010

Revisit search-based test generation

- *Insight: Mutants simulate various modifications to a program*
- **Mutation coverage** provides an ideal measurement of a test suite's capability to detect regression faults introduced by code changes
- That is, a test suite that kills more mutants is more effective in detecting regression faults



Hurdle - Recall our previous discussion ...



Chicken and egg?

- 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( ... );  
}
```

Hurdle - Revisit test generation strategies

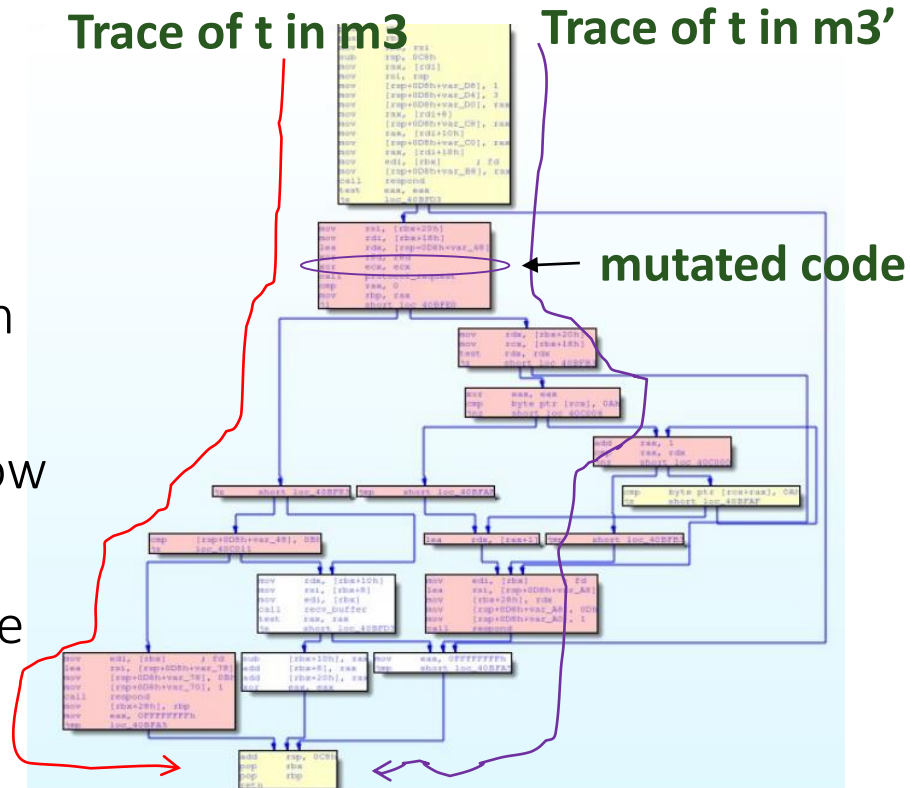
Goal: To generate a test + killing $m3'$

Test t : $m1; m2; m3'; \dots$ // $m3'$ is a mutant of $m3$

- Difficult to generate a test using line or branch coverage to kill the mutant
 - EITHER the test does not reach the minor changes in a mutant (e.g., the modified code in method $m3'$)
 - OR the test does not lead to a different output (e.g., the outputs of $m3$ and $m3'$ are different)

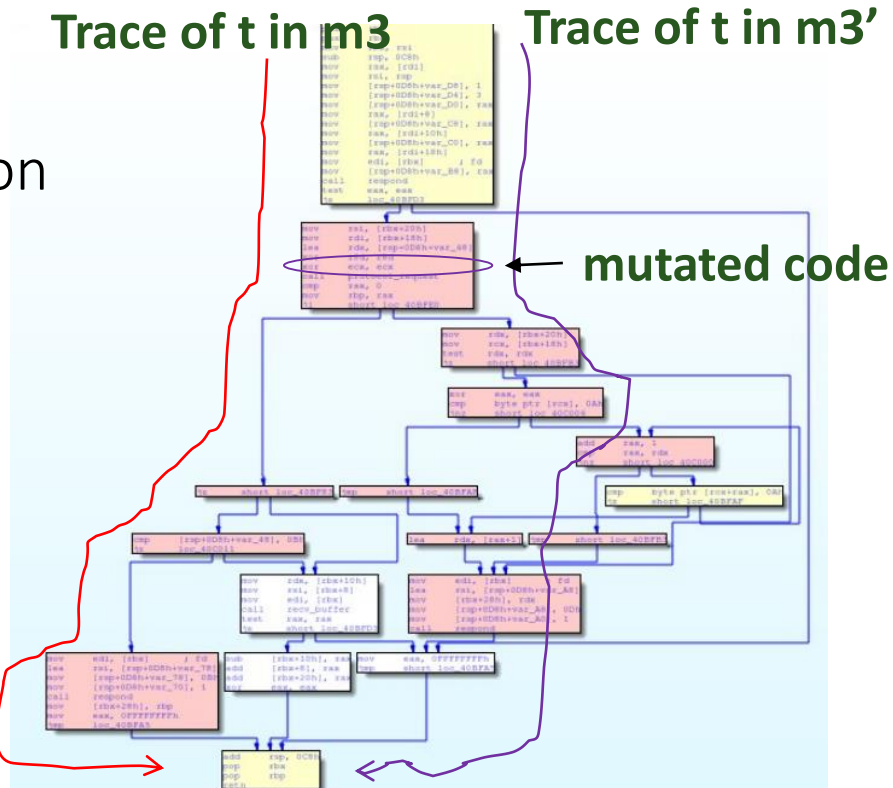
Fitness function for regression testing

- An intuitive solution is to use mutation score in the fitness function directly
- Issue 1: Killing mutants depend on assertions
- Issue 2: Mutation score is often low or even zero
- Not suitable to use mutation score directly in the fitness function



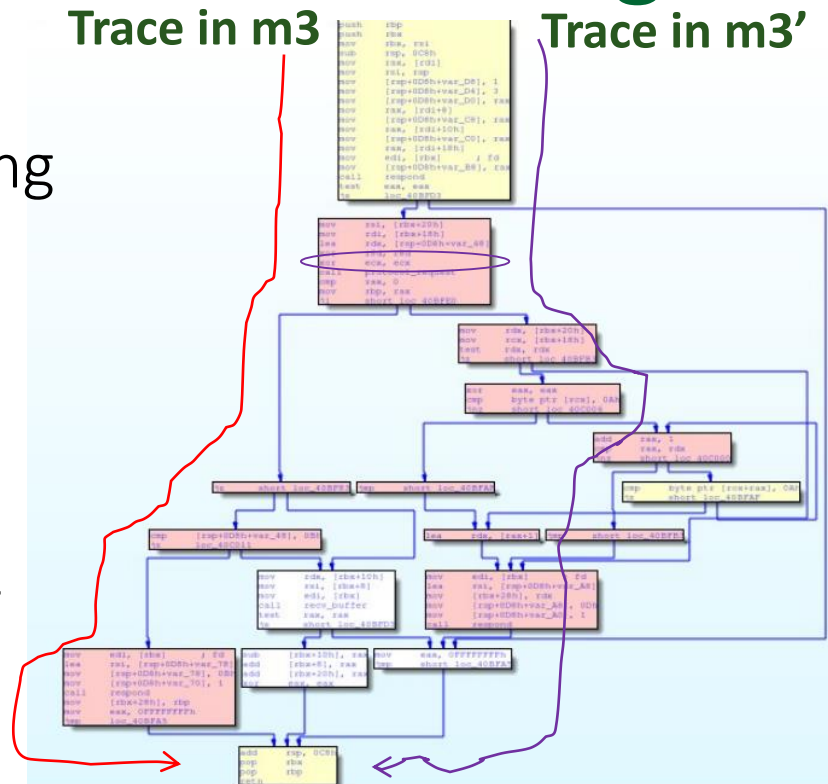
Fitness function for regression testing

- Can we have a fitness function that works without depending on mutation score?
- Idea: Favor tests that maximize the coverage of different code between m_3 and m_3'
- *Insight: Larger difference → Larger chance of killing m_3'*
- No longer depends on test oracles



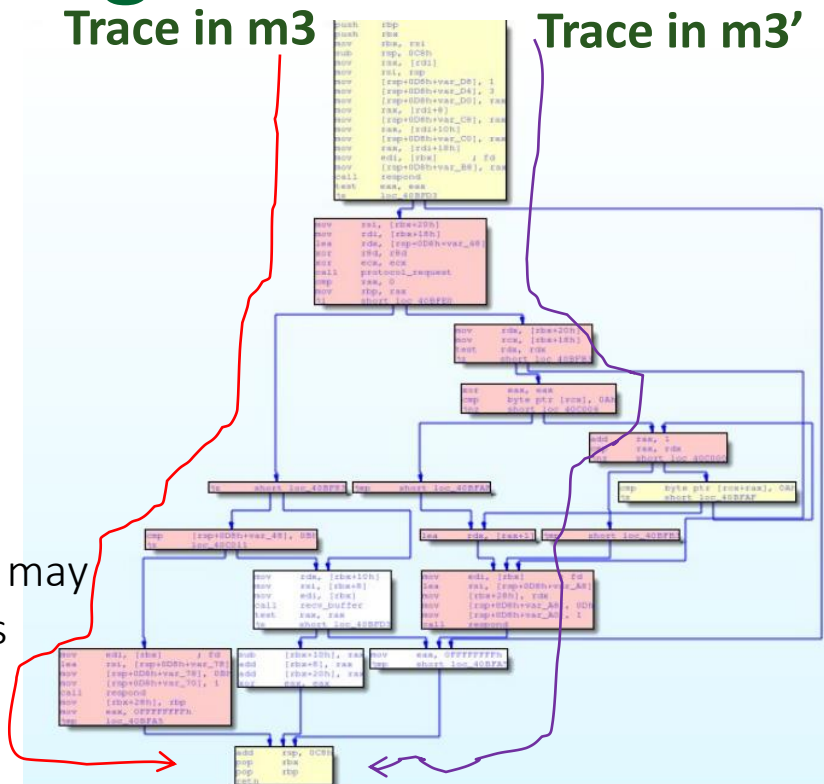
Fitness function based on coverage diff

- $t: m_1; m_2; m_3; m_4; m_5; \dots$
- Define fitness of t by summing its coverage diff between
 - $m_1; m_2; m_3; m_4; m_5; \dots$
 - $m_1, m_2, m_3', m_4, m_5; \dots$
- Aggregate the fitness values for all tests in a test suite for all mutants



Enhanced test oracle generation

- t: m1; m2; m3; m4; m5; ...
 - A test in the final test suite
- Two questions
 - What assertions
 - Where should we put these assertions
 - Test t with appropriate assertions may also kill mutants of other methods
 - After which method(s) should we insert assertions?



Five types of assertion candidates

- Primitive assertions ← Evosuite and Randoop default
 - Make assumptions on primitive method return values

```
DurationField var0 =  
    MillisDurationField.INSTANCE;  
long var1 = 43;  
long var2 = var0.subtract(var1, var1);  
assertEquals(var2, 0);
```

Execute the test, capture the resulting value of each primitive variable, and turn it into an assertion

Five types of assertion candidates

■ Comparison assertions

- ❑ Compare each pair of variables that reference objects of the same class

```
DateTime var0 = new DateTime();  
Chronology var1 = Chronology.getCopticUTC();  
DateTime var2 = var0.toDateTime(var1);  
→ assertFalse(var2.equals(var0));
```

var0 and var2 have the same type. Execute the test, turn their comparison result into an assertion

Five types of assertion candidates

■ Inspector assertions

- ❑ Call getters that return a primitive value to identify object states

```
long var0 = 38;  
Instant var1 = new Instant(var0);  
Instant var2 = var1.plus(var0);  
assertEquals(var2.getMillis(), 76);
```

→ Execute the test, record the return value of var2's getter and turn the value into an assertion

Five types of assertion candidates

■ Field assertions

- ❑ Execute the test
- ❑ Record the values of the last manipulated object's primitive fields
- ❑ Turn the recorded values into assertions

Five types of assertion candidates

■ String assertions

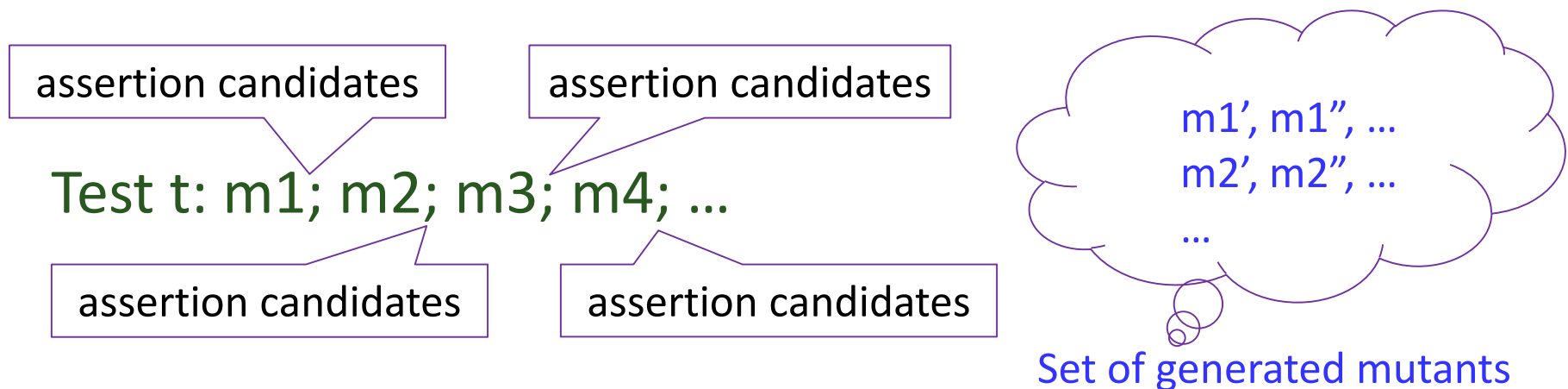
- ❑ Execute the test
- ❑ Record the string value of the last manipulated object
- ❑ Turn the recorded string value into assertions

```
int var0 = 7;  
Period var1 = Period.weeks(var0);  
assertEquals("P7W", var1.toString());
```

Assertion selection and placement

- We can generate many assertion candidates for a test
 - Not all of them make sense
- Questions
 - Which assertions should we use?
 - Where should we put these assertions?

Assertion selection and placement



- Optimization problem: select a minimal set of the assertion candidates at each execution point so that they collectively kill the maximum number of mutants

Assertion selection and placement

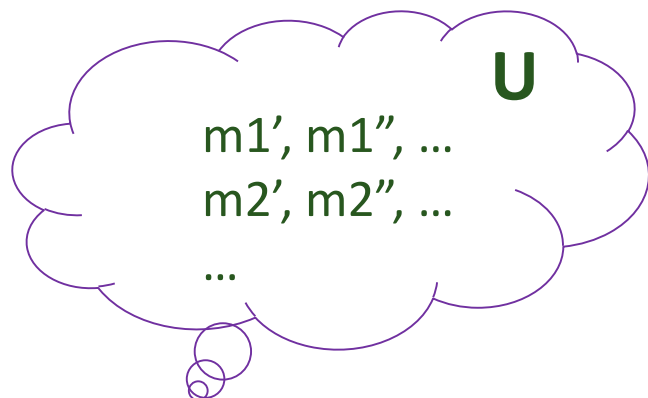
assertion candidates

assertion candidates

Test t: m1; m2; m3; m4; ...

assertion candidates

assertion candidates



U: Set of generated mutants

Greedy Strategy

1. Initialize U with the set of all mutants to be killed
2. Select an assertion that kills the greatest number of mutants in U
3. If multiple assertions kills the same number of mutants, randomly select one
4. Remove the killed mutants from U
5. Repeat steps 2 to 4 until no more mutants in U can be killed

Case study on two popular projects

Case Study	Classes	by developers		Mutants
		Unit Tests		
Joda-Time	123/220	3,493	14,778/23,145	
Commons-Math	220/413	1,739	25,226/43,273	

Case study subjects (selected/total)

Note:

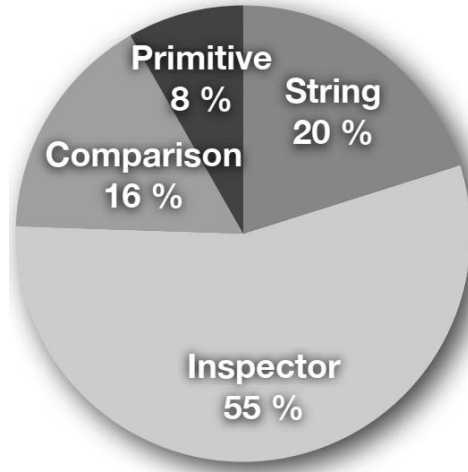
- Selected classes are those bundled with unit tests
- Selected mutants are those killed by bundled tests or generated tests for the selected classes

Case study on two popular projects

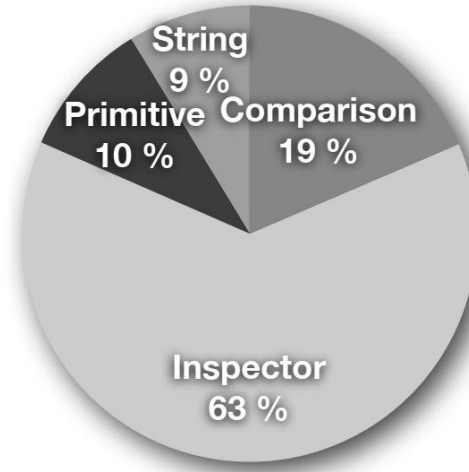
Case Study	Test cases	Average per test case		
		Lines	Assert.	Mut.
Joda-Time (man.)	3,493	4.89	4.55	3.14
Joda-Time (μ TEST)	1,268	8.48	3.30	9.67
Commons-Math (man.)	1,739	7.97	3.41	5.98
Commons-Math (μ TEST)	2,706	13.29	2.93	5.46

Manual vs generated test cases

Case study on two popular projects



(a) Joda-Time



(b) Commons-Math

Distribution of assertion types among generated tests

Case study on two popular projects

	Case Study	Manual	μ TEST	Σ
Total	Joda-Time	74.26%	82.95%	90.34%
	Commons-Math	41.25%	58.61%	61.15%
Avg./Unit	Joda-Time	70.36%	80.36%	88.35%
	Commons-Math	44.50%	65.87%	67.72%

Manual vs generated tests: Mutation scores

Experimental observation: μ TEST generates test suites and oracles that find significantly more seeded defects than manually written test suites

Question

- Is the test generation technique restricted to the detection of regression faults?

No if the faults can be detected by implicit oracles such as program crashes or exceptions

Further readings

- Wei Jin, Alessandro Orso, Tao Xie. Automated Behavioral Regression Testing, ICST 2010, pp. 137-146.
- Suzette Person, Guowei Yang, Neha Rungta, Sarfraz Khurshid. Directed Incremental Symbolic Execution, PLDI'11, pp. 504-515.
- S. Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey, STVR 2012, vol. 22, pp. 67-120.
- Lingming Zhang, Miryung Kim, Sarfraz Khurshid. FaultTracer: a spectrum-based approach to localizing failure-inducing program edits, Journal of Software, Evolution and Process, vol. 25 (12), 2013, pp. 1357-1383.
- Gordon Fraser and Andreas Zeller. Mutation-driven Generation of Unit Tests and Oracles, ISSTA 2010.

Further readings

- Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao: Chapter One - A Survey on Regression Test-Case Prioritization. Advances in Computers 113: 1-46 (2019)
- Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, Lu Zhang: Optimizing test prioritization via test distribution analysis. ESEC/SIGSOFT FSE 2018: 656-667
- Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, Lu Zhang: How does regression test prioritization perform in real-world software evolution? ICSE 2016: 535-546
- Regression Testing in Twitter products
 - <https://www.youtube.com/watch?v=2zjhKmV0UFA> (18:19 min)