Testing OO Programs

- **Testing**
  - **Black box testing**
  - **White box testing**
    - **Coverage metrics**
  - **Dataflow testing**
    - **Coverage metrics**
- **Class testing**
- **Polymorphism testing**

Testing

- **Black box testing**
  - **Tests functional specs of system**
    - **Test as a user**
    - **Does not need knowledge of system internals, just the API**
- **White box testing**
  - **Tests internal logic and value flow through system**
    - **Test with the code**
    - **Gain confidence through coverage metrics measure execution through parts of the code**

Coverage Metrics

- Control flow metrics
  - Branch coverage
  - Statement coverage
- Dataflow metrics
  - Def-use relations coverage
  - Seminal work by Elaine Weyuker and her students in defining metrics and showing their relation to one another

S. Rapps, E. Weyuker, "Selecting Software Test Data Using Data Flow Information, IEEE TSE, April 1985, pp 367-375.

Coverage Metrics

- Direct the selection of test data to make the testing procedure satisfy the metric
- Best criteria (?): *all-paths*
  - Select data that traverses all paths in a program
    - Data causing execution to traverse path p1 may not reveal an error on that path
    - There may be an infinite number of paths due to loops
- Rapps-Weyuker contribution
  - Designed a family of test data selection criteria so finite number of paths traversed
  - Systematic exploration of satisfying the criteria
  - Coverage criteria can be automatically checked

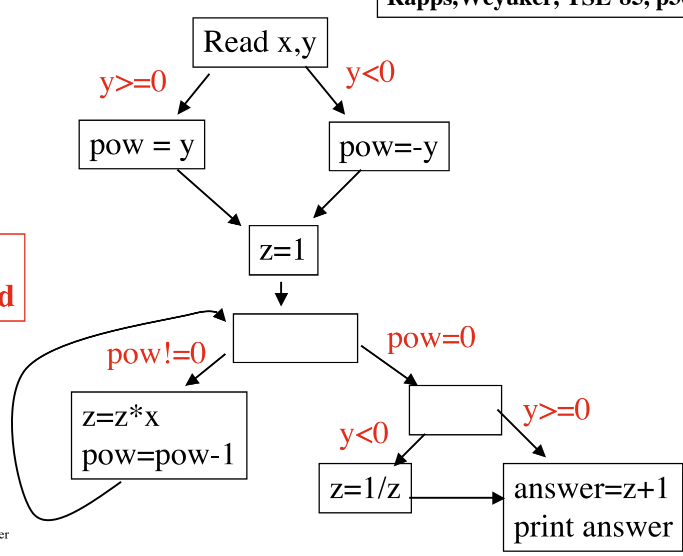
Definition-use graph

- Each variable occurrence is *definition, computation-use(c-use)* or *predicate-use(p-use)*
- Use the def-use associations of standard dataflow analysis, but differentiate the types of uses
  - Associate predicate uses with edges in the graph

Example

Rapps,Weyuker, TSE'85, p369

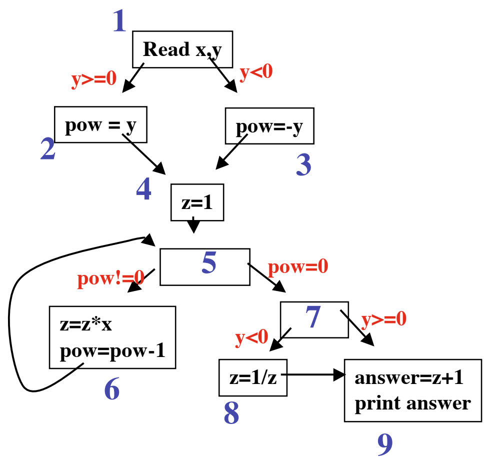
p-uses on edges in red



Example

Rapps,Weyuker, TSE'85, p369

node	c-use	def	edge	p-use
1	Φ	{x,y}	(1,2)	{y}
2	{y}	{pow}	(1,3)	{y}
3	{y}	{pow}	(5,6)	{pow}
4	Φ	{z}	(5,7)	{pow}
5	Φ	Φ	(7,8)	{y}
6	{x,z, pow}	{z, pow}	(7,9)	{y}
7	Φ	Φ		
8	{z}	{z}		
9	{z}	Φ		



Coverage Criteria

- **G is def-use graph and P is set of complete paths in G**
  - **P satisfies the *all-nodes* criterion if every node in G is included in P**
  - **P satisfies the *all-edges* criterion if every edge in G is included in P**
  - **P satisfies the *all-p-uses* criterion, if for every node j and every variable x defined at j, P includes a def-clear path wrt x from j to all p-uses of x reachable from j**

Coverage Criteria

- G is def-use graph and P is set of complete paths in G
  - P satisfies *all-c-uses/some-p-uses* criterion if for every node j and every x defined at j, P includes some def-clear path wrt x from j to all c-uses of x. if there are no such c-uses of x, then P includes some def-clear path wrt x from j to some edge contained in the set of p-uses of that def of x
  - Similar defn for *all-p-uses/some-c-uses*
  - P satisfies *all-uses* criterion if for every defn of variable x all of its p-uses and c-uses are covered

Coverage Criteria

- G is def-use graph and P is set of complete paths in G
  - P satisfies *all-du-paths* criterion if for every defn of variable x, all its du-paths are included (even multiple paths)
  - P satisfies *all-paths* criterion if P includes every complete path of G
    - There may be infinitely many such paths

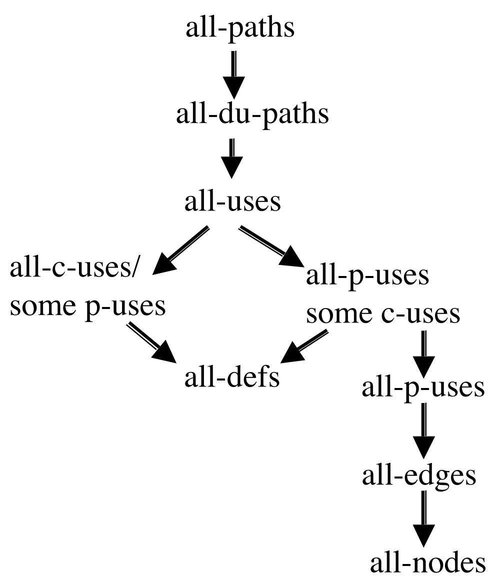
Criteria Selection

- **Tradeoff between strength of the criterion and how carefully the program is examined**
- **Weak criteria: *all-nodes* (statement coverage) and *all-edges* (branch coverage)**
- **How about *all-defs*?**
- **Need to compare criteria and to know which ones imply others**

Comparing Criteria

- **Criterion c1 *includes* criterion c2, if for every def-use graph G, any set of complete paths of G that satisfies c1 also satisfies c2.**
- **Criterion c1 *strictly includes* criterion c2, if c1 includes c2 and for some def-use graph G, there is a set of complete paths of g that satisfies c2 but not c1, c1 $\Rightarrow$ c2**
- **Criteria c1 and c2 are incomparable, if neither c1 $\Rightarrow$ c2 nor c2 $\Rightarrow$ c1.**

Criteria Inclusion Hierarchy



Testing of Classes

- **Claim class is basic unit of testing in OOPLs**
- **First approach for dataflow testing of classes (in C++)**
  - **Needs def-use analysis for object fields**
  - **Class call graph**
    - **Shows how methods in the class call each other**
    - **Includes incoming edges (from outside the class) to all public methods of the class**

M.J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes", FSE'94, pp 154-163.

Testing of Classes

- **Levels of testing**
  - ***intra-method*: unit testing**
  - ***inter-method*: tests a public method together with all methods reachable from it**
  - ***intra-class:* tests all possible interactions between public methods of a class (accessible by clients in arbitrary order)**
- **Previous techniques focused on *intra-class* testing**
- **Their approach: test def-use pairs**

Def-use pairs

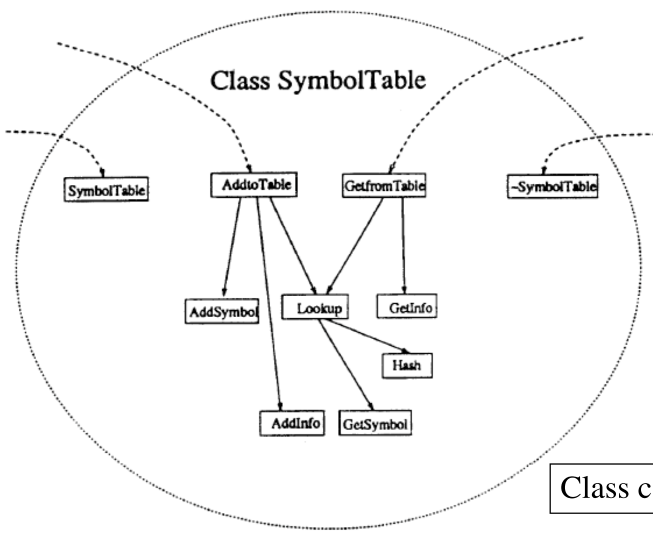
- ***Intra-method* def-use pair: both def and use within same method**
- ***Inter-method* def-use pair: scope of the pair is across more than one method frame**
- ***Intra-class* def-use pair: scope of the pair is across at least two public methods of the class and both the def and the use are in such methods**

Intra-class def-use pairs

- **Define data structure, class control flowgraph**
  - **Assume class call graph shows all possible calls between methods in the class and class entry points**
  - **Built from the class call graph by**
    - **Adding a driver that can call any public method in class**
    - **Expanding call graph nodes into control flow graphs for the corresponding methods**

Example

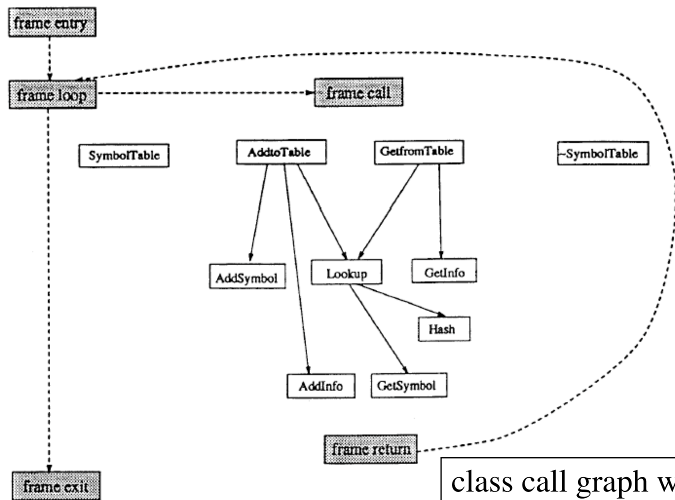
M.J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes", FSE'94, pp 154-163.



Class call graph

Example

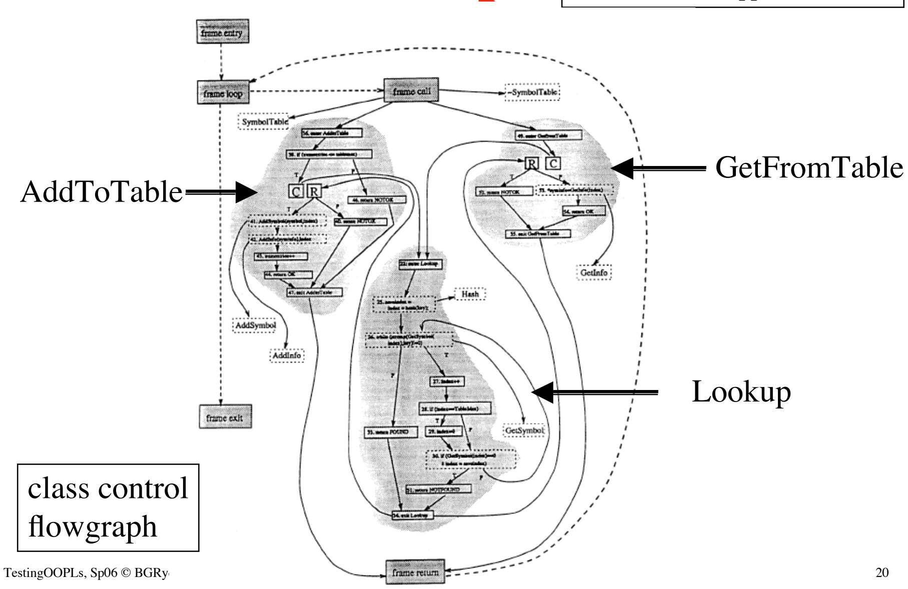
M.J. Harrold and G. Rothermel, “Performing Data Flow Testing on Classes”, FSE’94, pp 154-163.



class call graph with frame driver

Example

M.J. Harrold and G. Rothermel, “Performing Data Flow Testing on Classes”, FSE’94, pp 154-163.



class control flowgraph

Problems

- Approach may miss effects of client-induced aliasing
- Doesn't include testing for integrating classes
  - *Inter-class* testing: tests def-uses where def in one class and use in another class
- Can we reuse def-use info in testing derived classes?
- This paper ignored dynamic dispatch and polymorphism
  - Need to test a method with all or some of its possible call targets

Testing Polymorphism

- Idea: to test polymorphic calls in a set of classes
  - Need to identify the polymorphic targets accurately
  - Need to do reference analysis of incomplete OO programs (*program fragment reference analysis)*
  - Need to record actual call edges exercised during testing

A. Rountev, A. Milanova, B.G. Ryder, "Fragment Class Analysis for Testing of Polymorphism in Java Software", ICSE 2003

Coverage Metrics

- *Receiver class* **criterion (RC) - exercise all possible receiver classes**
- *Target method* **criterion (TM) - exercise all possible target methods**
- **Key idea: absolute (not relative) precision of reference analysis is important!**
  - **Uncovered call edges need to be checked by humans - very costly**

Assumptions

- **Select a set of classes *Cls* and set of methods and fields *Int* from *Cls***
- **Test suite for *Int* calls methods and fields from *Int* and does not access any methods/fields from *Cls* that are not in *Int***
- **Let *AllSuites(Int)* be the set of all possible test suites for *Int* (an infinite set)**
- **Tool input: *Cls, Int, T ∈ AllSuites(Int)***
- **Tool output: coverage of polymorphic call sites achieved by *T* wrt RC and TM criteria**

Testing Tool

1. Analysis component - computes RC and TM criteria
2. Instrumentation component - adds code to record call site targets and receiver types
3. Test harness - runs *T*
4. Reporting component - calculates coverage at call sites

Fragment Reference Analysis

- Nasko Rountev's PhD thesis (Rutgers 2002) developed framework for program fragment analyses
- Approach in ICSE'03 paper works for flow-insensitive reference analyses
- Idea: create a placeholder program to represent $T \in$ *AllSuites(Int)*
  - Code contains placeholder variables and statements that represent the unknown code
  - During analysis, the placeholders simulate possible effects of unknown code in an arbitrary test suite
  - Placeholder code plus *Cls* is analyzed by a whole-program reference analysis

Placeholder Code

```
main(){
  X ph_X;//one Ph_X for each class X in Cls
  ph_X = new X();//for each class X with
        //constructor in Int
//for every field f in Int declared in class X
//of type Y
  ph_Y = ph_X.f; ph_X.f = ph_Y;
//for each method in Int declared in class X
  ph_W = ph_X.m(ph_Y,…,ph_Z);
//for every subclass Y of class X
  ph_X = ph_Y;  ph_Y = (Y)ph_X;
}
```



```
package station;
public abstract class Link {
  public abstract void
      transmit(String message); }
class NormalLink extends Link { ...  }
class PriorityLink extends Link { ...  }
class SecureLink extends Link { ...  }
class LoggingLink extends Link { ...  }

public class Station {
  private Link link = new NormalLink();
  private int msg_id = 0;
  public void sendMessage(String m) {
c1: link.transmit(msg_id++ + " " + m);
    if (msg_id==10)
      link = new PriorityLink(); }
  public void report(Link l)
    { c2: l.transmit("id = " + msg_id);} }

public class Factory {
  private boolean secure = false;
  public Link getLink() {
    if (secure) return new SecureLink();
    else        return new NormalLink(); }
  public void makeSecure()
        { secure = true; } }
```

Example of package to be tested; arrows point to methods in *Int;* also need constructors of Factory, Station

```
import station;
main() {
        Station ph_Station;
        Factory ph_Factory;
        Link ph_Link;
        String ph_String;
        ph_Station = new Station();
        ph_Factory = new Factory();
        ph_String = new String();
        ph_Station.sendMessage(ph_String);
        ph_Station.report(ph_Link);
        ph_Link = ph_Factory.getLink();
        ph_Factory.makeSecure();
        ph_Link.transmit(ph_String);
}
```

Placeholder code corresponding to example on previous slide.

Experiments

- **Used java.util.zip, java.text, com.lowagie.text libraries as data to be tested**
  - **Selected specific testing tasks for each library**
- **For each testing task, found CIs and number of polymorphic call sites**
- **Tested 3 fragment reference analyses: RTA_f And_f $0\text{-}CFA_f$**

Description of Data

Task	Package	Functionality	#Classes	#PolySites
task1	`java.text`	boundaries in text	12	12
task2	`java.text`	formatting of numbers/dates	13	79
task3	`java.text`	text collation	12	2
task4	`java.util.zip`	ZIP files	8	5
task5	`java.util.zip`	ZIP output streams	8	18
task6	`gnu.math`	complex numbers	8	194
task7	`com.lowagie.text`	paragraphs in PDF docs	24	199
task8	`com.lowagie.text`	lists in PDF docs	24	169

Table 2. Description of testing tasks.

Findings

Task	Hierarchy		RTA_f		$0\text{-}CFA_f$		AND_f	
	C_{RC}	C_{TM}	C_{RC}	C_{TM}	C_{RC}	C_{TM}	C_{RC}	C_{TM}
task1	100%	100%	100%	100%	100%	100%	100%	100%
task2	67%	63%	67%	63%	76%	72%	76%	72%
task3	50%	100%	50%	100%	100%	100%	100%	100%
task4	31%	63%	45%	71%	100%	100%	100%	100%
task5	18%	21%	88%	92%	100%	100%	100%	100%
task6	76%	85%	76%	85%	97%	98%	98%	98%
task7	10%	15%	32%	48%	82%	93%	87%	93%
task8	5%	9%	18%	29%	62%	62%	62%	62%

Conclusions

- CHA and RTA_f compute significant numbers of infeasible receiver classes
- $0\text{-}CFA_f$ and And_f perform well; achieve perfect precision in over half the cases!
  - Practical cost: in all cases under 20 seconds
- First study of absolute precision of reference analyses