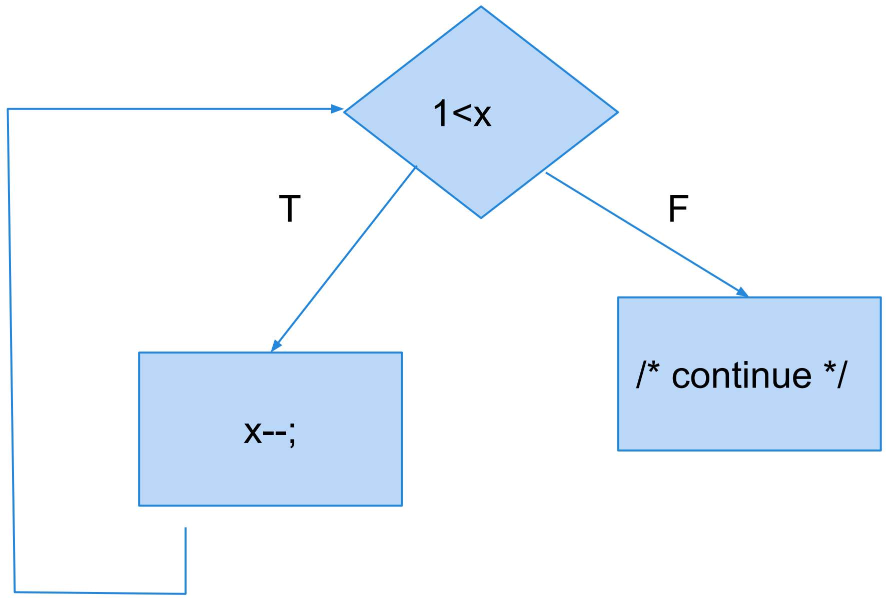
Data Flow Testing

CSCE 747 - Lecture 10 - 02/20/2018

Control Flow

- Capture dependencies in terms of how control passes between parts of a program.
- We care about the effect of a statement when it affects the path taken.
  - but deemphasize the information being transmitted.



Data Flow

- Another view - program statements compute and transform data…
  - So, look at how that data is passed through the program.
- Reason about **data** dependence
  - A variable is used here - where does its value come from?
  - Is this value ever used?
  - Is this variable properly initialized?
  - If the expression assigned to a variable is changed what else would be affected?

Data Flow Analyses

- Used to detect faults and other anomalies.

	Any-Paths	**All-Paths**
Forward (pred)	**Reach** *U* may be preceded by G without an intervening *K*	**Avail** *U* is always preceded by G without an intervening *K*
Backward (succ)	**Live** *D* may lead to *G* before *K*	**Inevitability** *D* always leads to *G* before *K*

- Also can be used to derive test cases.
  - Have we covered the data dependencies?

Variable Aliasing

Dealing With Arrays/Pointers

- Arrays and pointers (including object references and arguments) introduce issues.
  - It is not possible to determine whether two access refer to the same storage location.
    - ```
      a[x] = 13;
      k = a[y];
      ```
      - Are these a def-use pair?
    - ```
      a[2] = 42;
      i = b[2];
      ```
      - Are these a def-use pair?
        - Aliasing = two names refer to the same memory location.

Aliasing

- *Aliasing* is when two names refer to the same memory location.
  - ```
    int[] a = new int[3];
    int[] b = a;
    a[2] = 42;
    i = b[2];
    ```
  - a and b are aliases.
- Worse in C:

```
p = &b;
*(p + i) = k;
```
```

Uncertainty

- Dynamic references and aliasing introduce uncertainty into data flow analysis.
    - Instead of a definition or use of one variable, may have a potential def or use of a set of variables.
- Proper treatment depends on purpose of analysis:
    - If we examine variable initialization, might not want to treat assignment to a potential alias as initialization.
    - May wish to treat a use of a potential alias of *v* as a use of *v*.

Dealing With Uncertainty

- Basic option: Treat all potential aliases as definitions and uses of the same variable:

```
a[1] = 13;
k = a[2];
```

Def of a[1], use of a[2].

```
a[x] = 13;
k = a[y];
```

Def and use of array a.

- Easiest and cheapest option when performing an analysis.
- Can be very imprecise.
  - They are only the same if x and y are the same.

Dealing With Uncertainty

- Treat uncertainty about aliases like uncertainty about control flow.

```
a[x] = 13;
k = a[y];
```

```
a[x] = 13;
if(x == y)    k = a[x];
else          k = a[y];
```

- In transformed code, all array references are distinct.
  - Any-path analysis - create a def-use pair, but assignment to a[y] does not erase definition to a[x].
  - Gen sets include everything that might be references, kill sets only include definite references.

Dealing With Uncertainty

```
a[x] = 13;
k = a[y];
```

```
a[x] = 13;
if(x == y)    k = a[x];
else          k = a[y];
```

- In transformed code, all array references are distinct.
  - Any-path analysis - create a def-use pair, but assignment to a[y] does not erase definition to a[x].
  - All-paths analysis - a definition to a[x] makes only that expression available. Assignment to a[y] kills a[x].
    - Gen sets should include only what is definitely referenced and kill sets should include all possible aliases.

Dealing With Nonlocal Information

- fromCust and toCust may be references to the same object.
  - from/toHome and from/toWork may also reference the same object.
- Common option - treat all nonlocal information as unknown.
  - Treat Customer/PhoneNum objects as potential aliases.
  - Be careful - may result in results so imprecise they are useless.

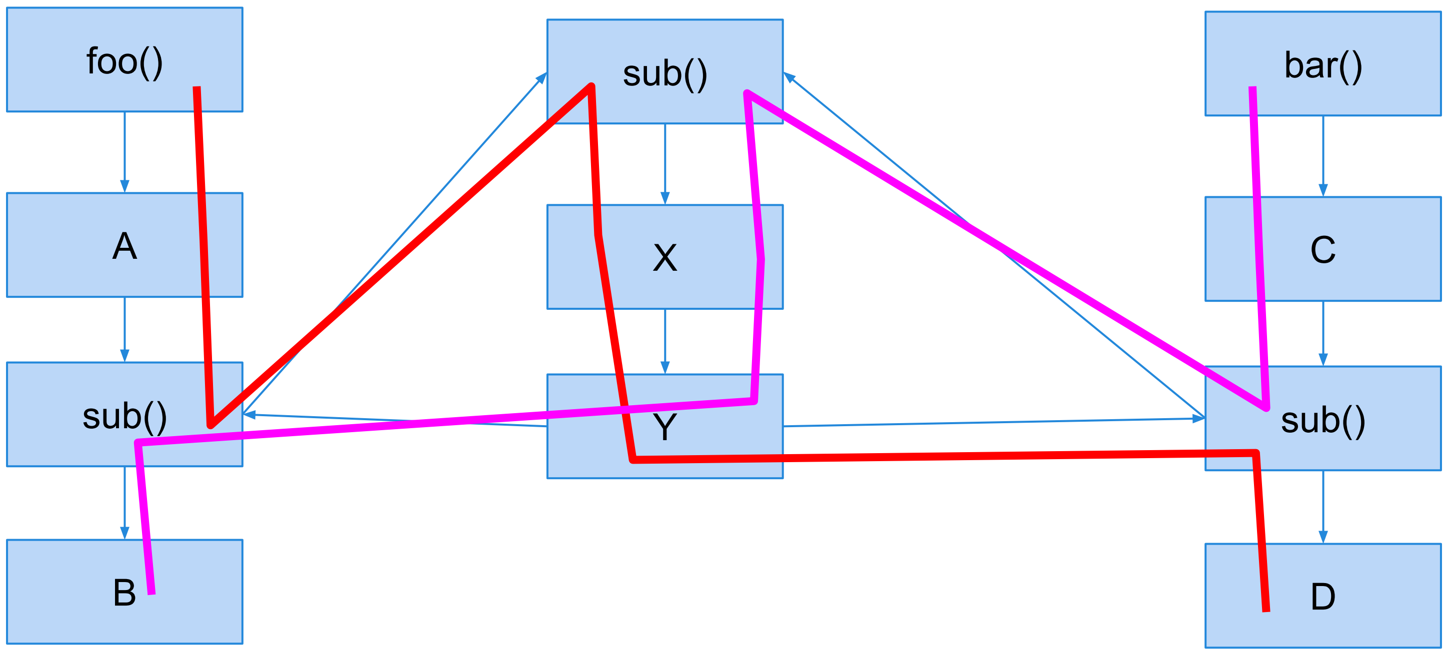
```
public void transfer(Customer fromCust,
Customer toCust){
      PhoneNum fromHome =
            fromCust.getHomePhone();
      PhoneNum fromWork =
            fromCust.getWorkPhone();
      PhoneNum toHome =
            toCust.getHomePhone();
      PhoneNum toWork =
            toCust.getWorkPhone();
}
```

Interprocedural Analysis

Interprocedural Analysis - Control Flow

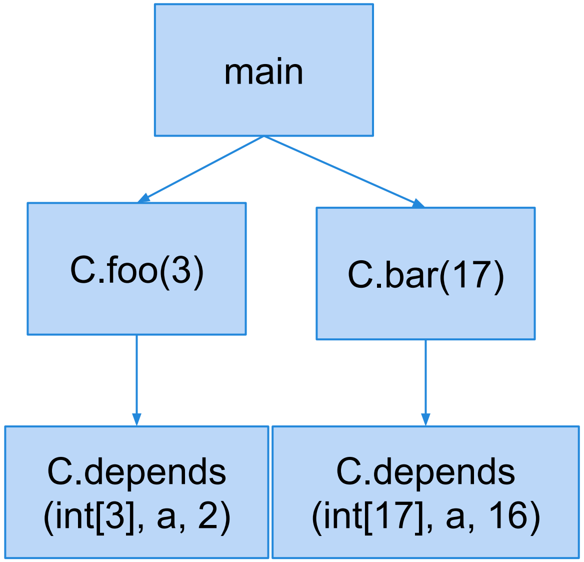
- First op… …dures in a large C…

Problem - infeasible paths!



Context-Sensitivity

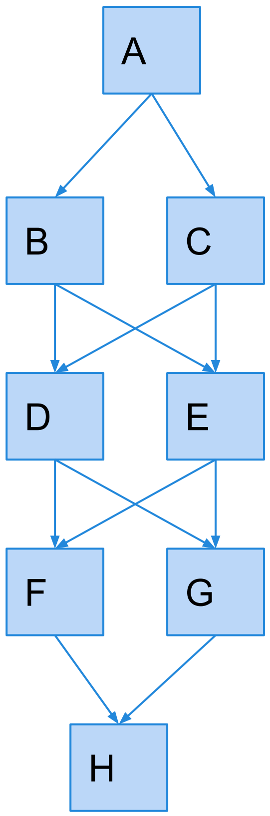
```
public class Context{
    public static void main(String args[]){
        Context c = new Context();
        c.foo(3);
        c.bar(17);
    }
    void foo(int n){
        int[] a = new int[n];
        depends(a,2);
    }
    void bar(int n){
        int[] a = new int[n];
        depends(a,16);
    }
    void depends(int[] a, int n){
        a[n] = 42;
    }
}
```



Context-Sensitive

Context-Sensitive Analysis

- Copy the called procedure for each point that it is called.
- Problem - the number of contexts a procedure is called in is exponentially higher than the number of procedures.
    - Precise, but expensive analysis.
- In practice, only feasible for small groups of related procedures.

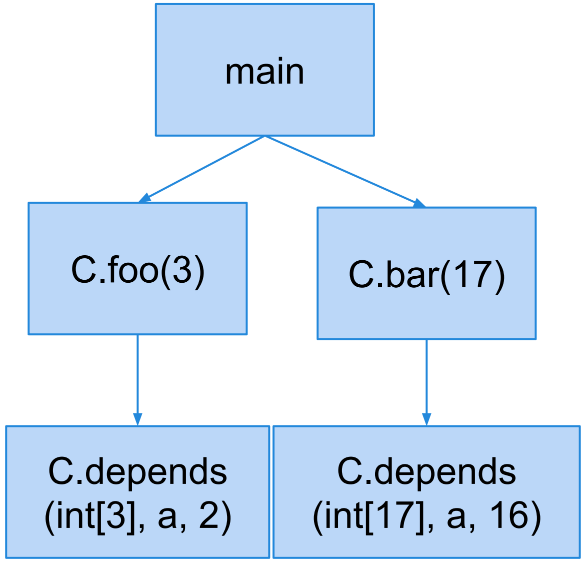


Context-Insensitive Analysis

- Unhandled exception analysis
  - If procedure A calls procedure B that throws an exception, A must handle or declare that exception.
  - Analysis steps hierarchically through the call graph.
- Two conditions:
  - Information needed to analyze calling procedure must be small.
  - Information about the called procedure must be independent of caller (context-insensitive)
- Analysis can start from leaves of call graph and work upward to the root.

Flow-Sensitivity

- Aliasing information requires context.
- Some analyses can sacrifice precision on another aspect: control-flow information
  - Call graphs are flow-insensitive.



Insensitive Pointer Analysis

- Treat each statement as a constraint.
  `x = y;` (where y is a pointer)
- Note that x may refer to any of the same objects that y refers to.
  - References(x) $\supseteq$ References(y) is a constraint independent of the path taken.
  - Procedure calls are assignments of values to arguments.
- Results are imprecise, but better than just assuming that any two pointers might refer to the same object.

Data Flow Testing

Overcoming Limitations of Path Coverage

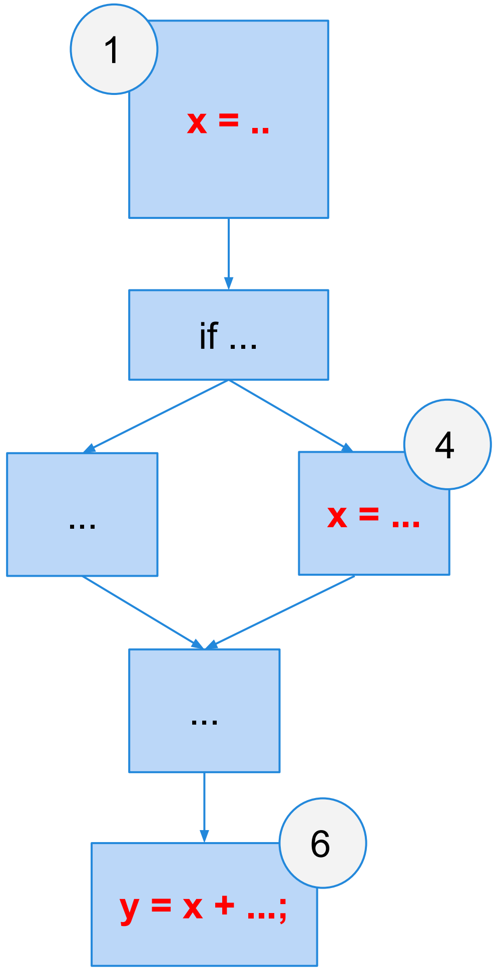
- We can potentially expose many faults by targeting particular *paths* of execution.
- Full path coverage is impossible.
- What are the important paths to cover?
    - Some methods impose heuristic limitations.
        - Loop boundary coverage
    - Can also use data flow information to select a subset of paths based on how one element can affect the computation of another.

Choosing the Paths

- Branch or MC/DC coverage already cover many paths. What are the remaining paths that are important to cover?
- Basis of data flow testing - computing the wrong value leads to a failure only when that value is *used*.
    - Pair definitions with usages.
    - Ensure that definitions are actually used.
    - Select a path where a fault is more likely to propagate to an observable failure.

Review - Def-Use Pairs

- Incorrect computation of x at either 1 or 4 could be revealed if used at 6.
- (1,6) and (4,6) are *DU pairs* for x.
  - DU Pair = there exists a *definition-clear path* between the definition of x and a use of x.
  - If x is redefined on the path, the original definition is *killed* and replaced.



Def-Use Pairs

- `++counter, counter++, counter+=1`
  `counter = counter + 1`
  - These are equivalent. They are a *use* of counter, then a new *definition* of counter.
- `char *ptr = *otherPtr`
  - Need a policy for how you deal with aliasing.
  - Ad-hoc option:
    - Definition of **string** *ptr
    - Use of **index** ptr, string *otherPtr, and index otherPtr.
- `ptr++`
  - Use of index ptr, and a definition of both the index and string *ptr.
  - Change to index moves the pointer to a new location.

All DU Pair Coverage

- Requires each DU pair be exercised in at least one program execution.
  - Erroneous values produced by one statement might be revealed if used in another statement.

$$\text{Coverage} = \frac{\text{number exercised DU pairs}}{\text{number of DU pairs}}$$

- Can easily achieve structural coverage without covering all DU pairs.

All DU Paths Coverage

- One DU pair might belong to many execution paths. Cover all simple (non-looping) paths at least once.
  - Can reveal faults where a path is exercised that should use a certain definition but doesn't.

$$\text{Coverage} = \frac{\text{number of exercised DU paths}}{\text{number of DU paths}}$$

Path Explosion Problem

- Even without looping paths, the number of SU paths can be exponential to the size of the program.
- When code between definition and use is irrelevant to that variable, but contains many control paths.

```
void countBits(char ch){
     int count = 0;
     if (ch & 1)   ++count;
     if (ch & 2)   ++count;
     if (ch & 4)   ++count;
     if (ch & 8)   ++count;
     if (ch & 16)  ++count;
     if (ch & 32)  ++count;
     if (ch & 64)  ++count;
     if (ch & 128) ++count;
     printf("'%c' (0X%02X) has %d bits set to 1\n", ch, ch, count);
}
```

All Definitions Coverage

- All DU Pairs/All DU Paths are powerful and often practical, but may be too expensive in some situations.
- In those cases, pair each definition with at least one use.

$$\text{Coverage} = \frac{\text{number of covered definitions}}{\text{number of definitions}}$$

Dealing With Aliasing

- Requires trade-off between precision and computational efficiency.
- Underestimate potential aliases
  - Could miss *def-use* pairs
- Overestimate potential aliases
  - Could have infeasible pairs, leading to unsatisfiable coverage obligations
- What is a suitable approximation of potential aliases for testing?

Infeasibility Problem

- Metrics may ask for impossible test cases.
- Path-based metrics aggravates the problem by requiring infeasible combinations of feasible elements.
  - Alias analysis may add additional infeasible paths.
- All Definitions Coverage and All DU-Pairs Coverage often reasonable.
  - All DU-Paths is much harder to fulfill.

Activity - DU Pairs

- Identify all DU pairs and write test cases to achieve All DU Pair Coverage.
  - Hint - remember that there is a loop.

```
int doSomething(int x, int y)
{
    while(y > 0) {
        if(x > 0) {
            y = y - x;
        }else {
            x = x + 1;
        }
    }
    return x + y;
}
```

Activity - DU Pairs

```
1.  int doSomething(int x, int y)
2.  {
3.      while(y > 0) {
4.          if(x > 0) {
5.              y = y - x;
6.          }else {
7.              x = x + 1;
8.          }
9.      }
10.     return x + y;
11. }
```

Variable	Defs	Uses
x	1, 7	4, 5, 7, 10
y	1, 5	3, 5, 10

Variable	D-U Pairs
x	(1, 4), (1, 5), (1, 7), (1, 10), (7, 4), (7, 5), (7, 7), (7, 10)
y	(1, 3), (1, 5), (1, 10), (5, 3), (5, 5), (5, 10)

Activity - DU Pairs

```
1. int doSomething(int x, int y)
2. {
3.      while(y > 0) {
4.          if(x > 0) {
5.              y = y - x;
6.          }else {
7.              x = x + 1;
8.          }
9.      }
10.     return x + y;
11. }
```

Variable	D-U Pairs
x	~~(1, 4)~~, ~~(1, 5)~~, ~~(1, 7)~~, ~~(1, 10)~~, ~~(7, 4)~~, ~~(7, 5)~~, ~~(7, 7)~~, ~~(7, 10)~~
y	~~(1, 3)~~, ~~(1, 5)~~, ~~(1, 10)~~, ~~(5, 3)~~, ~~(5, 5)~~, ~~(5, 10)~~

Test 1: (x = 1, y = 2)
Covers lines 1, 3, 4, 5, 3, 4, 5, 3, 10

Test 2: (x = -1, y = 1)
Covers lines 1, 3, 4, 6, 7, 3, 4, 6, 7, 3, 4, 5, 3, 10

Test 3: (x = 1, y = 0)
Covers lines 1, 3, 8

We Have Learned

- Arrays, pointers, and complex data structures introduce uncertainty into analysis.
  - Requires a policy for how aliasing is handled.
  - Trade-off between computational feasibility and precision.
- Analyses must handle non-local references.
  - Similar trade-off. Can gain efficiency by sacrificing flow sensitivity and context sensitivity.

We Have Learned

- If there is a fault in a computation, we can observe it by looking at where the computation is used.
- By identifying DU pairs and paths, we can create tests that trigger faults along those paths.
    - All DU Pairs coverage
    - All DU Paths coverage
    - All Definitions coverage

Next Class

- Model-Based Testing

- Reading: Chapter 14
- Homework:
  - Homework 2 is out - Due March 6th
  - Reading Assignment 2 due tonight