Functionality Debugging

CS 4272

Abhik Roychoudhury
National University of Singapore

Acknowledgment: An earlier Guest Lecture by Tao Wang

The context

- System Modeling
  - UML, Statecharts
- Partitioning
  - ILP based methods are a candidate
- Scheduling
  - Many choices --- aperiodic, periodic
- Implementations

Under Implementation

- Many aspects
  - Functionality Validation
  - Timing Analysis and Validation
  - Choice of Platforms
  - Compilation and compiler-guided platform customization.
- Today's lecture
  - Functionality Validation

Outline of Material on Debugging

- Introduction
- Program Slicing
- Testing based Fault Localization
- Summary

Introduction

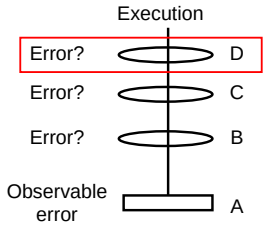
- Software Debugging
  - Time consuming
- Automatic methods for syntax errors

Semantic errors?

```
String s1 =…
String s2 =…
…
If (s1 == s2)
    ……
```

Software Debugging

- Typical Debugging Steps



1. Hypothesize the cause of the error
2. Try to confirm it

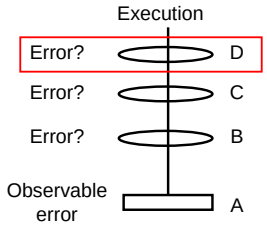
Software Debugging

1. Hypothesize the cause of the error
   - Program analysis
   - Identify suspicious statements

2. Try to confirm it

Software Debugging

- Typical Debugging Steps



1. Hypothesize the cause of the error
2. Try to confirm it

Software Debugging

1. Hypothesize the cause of the error
   - Program analysis
   - Identify suspicious statements

2. Try to confirm it
   - Collect information to help programmers

Program Analysis for Debugging

- What to analyze?
  - the execution run, Dynamic Analysis
  - the source code, Static Analysis
- How to analyze?
  - Program Slicing
  - Testing based Fault Localization

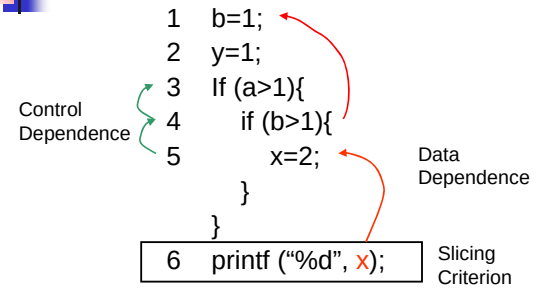
Outline

- Introduction
- Program Slicing
- Testing based Fault Localization
- Summary

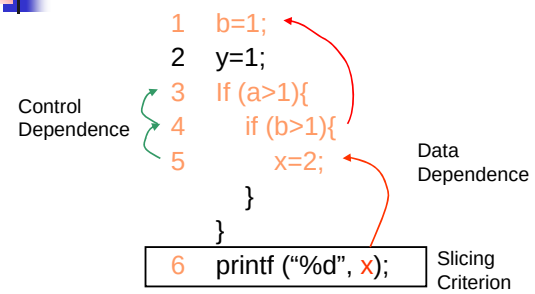
Program Slicing

```
1   b=1;
2   y=1;
3   If (a>1){
4       if (b>1){
5           x=2;
        }
    }
6   printf ("%d", x);
```

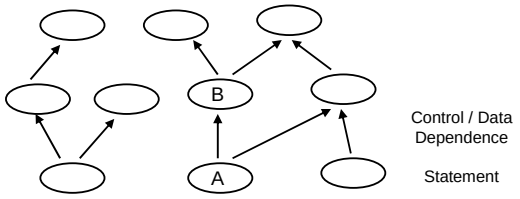
Program Slicing



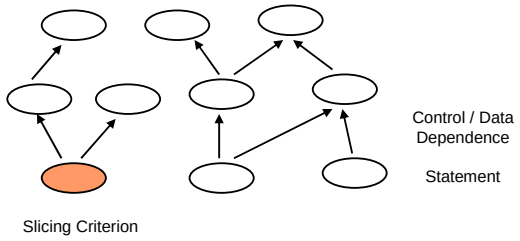
Program Slicing



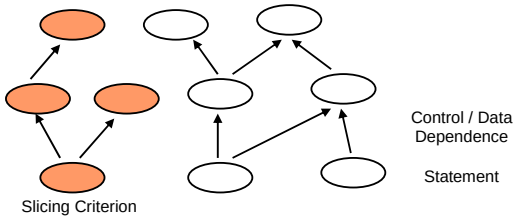
Dependence Graph



Program Slicing



Program Slicing



Static vs Dynamic Slicing

- Static Slicing
  - source code
  - statement
  - static dependence
- Dynamic Slicing
  - a particular execution
  - statement instance
  - dynamic dependence
  - Corresponds naturally to debugging via testing.

Static vs Dynamic Slicing

```
1  b=1;
2  If (a>1)
3     x=1;
4  else
5     x=2;
6  printf ("%d", x);
```

Slicing Criterion

Static vs Dynamic Slicing

```
1  p.f = 1;
2  x= q.f;
3  printf ("%d", x);
```

Slicing Criterion

p and q point to the same object?

- Static points-to analysis is always conservative

Program Slice

Static	Dynamic		
1		1	b=1; input: a=2
2	2	2	x=1;
3		3	If (a>1){
4		4	if (b>1){
5		5	x=2;
			}
			}
6	6	6	printf ("%d", x);

Formalizing Slicing

- Dependence Graphs
  - Nodes are statements for Static Slicing
  - Edges are data or control dependencies
- For dynamic slicing
  - The nodes are statement instances.
    - A statement can be executed many times in a trace, each execution is a statement instance.
  - Edges are control/data dep. as before.

Dynamic Dependence Graph

- G = (V, E)
  - V = All statement occurrences for the test input under consideration.
  - E = Data and Control dependences between statement occurrences.
- Slicing Criterion
  - A node in the DDG
- Slice computation
  - Nodes reachable from slicing criterion

Dynamic Data dependences

```
V := 1;
...
U := V
```

An edge from a variable usage to the latest definition of the variable.

→ Do we consider this data dependence edge ?

```
A[i] := 1;
...
U := A[j]
```

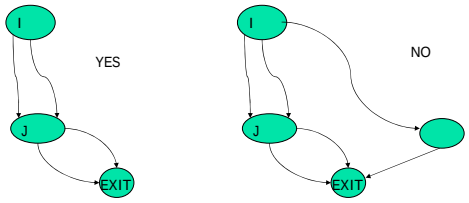
→ Remember that the slicing is for an input, so the addresses are resolved

→ We thus define data dependences corresponding to memory locations rather than variable names.

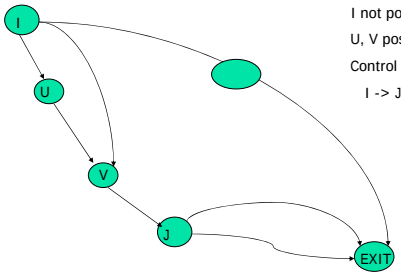
Control Dependences

Post-dominated: I,J – nodes in Control Flow Graph

I is post-dominated by J iff all paths from I to EXIT pass through J



Control Dependences



I not post-dom by J

U, V post-dom by J

Control dependence

I -> J

Dynamic Slice Illustration

```
0   scanf("%d", &A);
1.  If (A == 0){;
2       W = X;
3       U = A;
4   }
5   printf("%d\n", U);
```

Criterion: 5,U with input A == 0

Trace: <0,1,2,3,4,5> Slice = {0,1,3,5}

Data dependences encountered 5 -> 3, 1 -> 0

Control dependence encountered 3 -> 1

Computation of dynamic slice

- Backward
  - Execute the program for input
  - Store the trace during execution
  - Traverse the trace from the end.
- Forward
  - Execute the program.
  - Keep track of slices for various possible criteria --- computes many slices.

Backward Slicing

Run the program for selected input.

Store execution trace.

Set slicing criterion.

Traverse the trace from the slicing criterion to the beginning of the trace by going through control / data dependencies.

Forward Dynamic Slicing

- No need to store trace.
- Execute the program for selected input.
- At every step, compute and update the slice (from among statements seen so far), for different slicing criteria
  - Various variables for the current line

Comparison

- Forward
  - No need to store trace
  - Store slices for many slicing criteria
- Backward
  - Store trace
  - Goal-directed, compute only the needed slice.

A real-life example

- Apache JMeter
  - Performance Testing tool
  - 43,400 Lines of Code (LoC)
  - 389 Java classes

Dynamic Slicing

```
1. void setRunningVersion(boolean runningVersion){

2.   if( runningVersion ) {
3.       savedValue = value;
     }
     else{
4.       savedValue = "";
     }

5    this.runningVersion = runningVersion;

6.   System.out.println(savedValue);
   }
```

a bug from JMeter should be *savedValue = NULL;*

Dynamic Slicing

```
1. void setRunningVersion(boolean runningVersion){

2.   if( runningVersion ) {
3.       savedValue = value;
     }
     else{
4.       savedValue = "";
     }

5    this.runningVersion = runningVersion;

6.   System.out.println(savedValue);
   }
```

runningVersion = false

null ?

Dynamic Slice

```
1. void setRunningVersion(boolean runningVersion)

2.   if( runningVersion ) {
3.       savedValue = value;
     }
     else{
4.       savedValue = "";
     }

5    this.runningVersion = runningVersion;

6.   System.out.println(savedValue);
   }
```

Slicing Criterion

Testing and Debugging

- Two very close software development activities

Testing Stage → Debugging Stage

Testing Methods

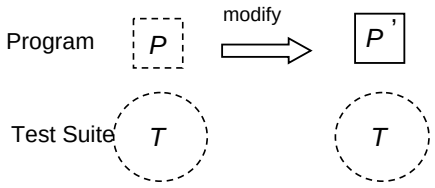
Debugging Methods

independent?

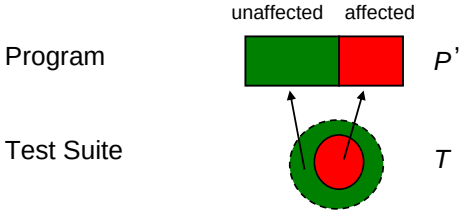
Testing and Debugging

- Testing and Debugging techniques can help each other
- Improve testing
  - Forward Slicing for Regression Testing

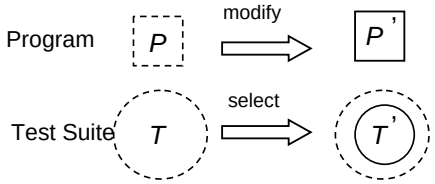
Regression Testing



Regression Testing



Regression Testing



Regression Testing

1. Identify a set of program components affected by the modified parts

Use Slicing

2. Identify *T'* of the *T* to test the affected components

Forward Slicing for Regression Testing

```
1   a=1;
2   if (b>1)
3      y=x+10;
4   b=2;
```

P

	Test 1, b=0	*Test 2, b=5*
1	1	1
2	2	2
3		3
4	4	4

Forward Slicing for Regression Testing

```
                          Test 1, b=0   Test 2, b=5
1   a=1;                      1             1
    x=1;
2   if (b>1)                  2             2
3      y=x+10;                              3
4   b=2;                      4             4
```

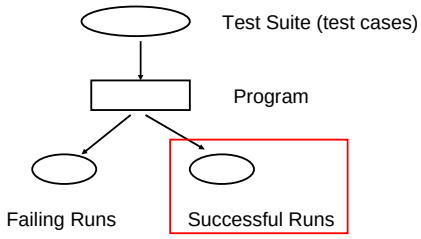
P'

Outline

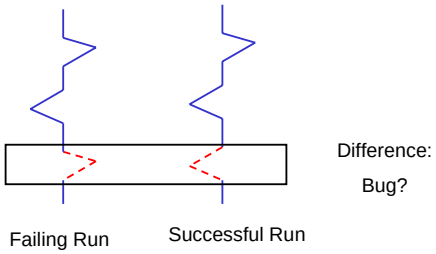
- Introduction
- Program Slicing
- Testing based Fault Localization
- Summary

Testing Based Fault Localization

- Use testing to help debugging



Testing Based Fault Localization



Testing Based Fault Localization

- What to Compare
  - choice of the Execution Run

Testing Based Fault Localization

- What to Compare
  - choice of the Execution Run
- How to Compare

Choice of the Execution Run

- Failing run
  - Select one failing run for comparison
  - Different failing runs may correspond to different error causes
- Successful Run
  - A Single successful run, VS
  - A Set of successful runs

Choice of a Single Successful Run

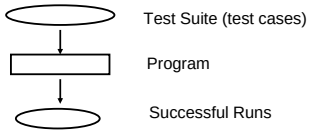
- The difference can be related to:
  1. different inputs
  2. the error
- The successful run and the failing run should be as similar as possible
  - Choose a “good” successful run for comparison

Testing Based Fault Localization

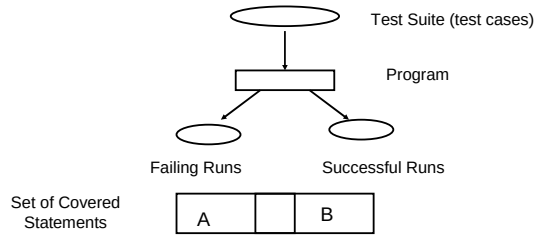
- What to Compare
  - choice of the Execution Run
- How to Compare
  - statements
  - branches
  - potential invariants
  - variable values

Fault Localization - statement

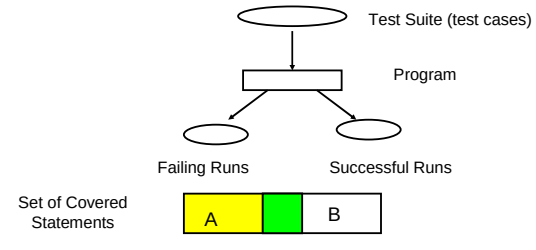
- Statement Coverage for Testing
  - every statement should be executed at least once with test cases in the Test Suite
  - intuition for this Coverage Criteria



Fault Localization - statement



Fault Localization - statement



Fault Localization - branches

```
1.  v=0;
2.  if (x>0)        →  if (x>=0)
3.      u=5;
4.  else
5.      u=v;
6.
    printf("%d",u);
```

Fault Localization - branches

Failing run, x=0

```
1.  v=0;
2.  if (x>0)
4.  else
5.      u=v;
6.
    printf("%d",u);
```

Successful run, x=1

```
1.  v=0;
2.  if (x>0)
3.      u=5;
6.
    printf("%d",u);
```

Compare Corresponding Statement Instances

```
1.  while (a){
2.      if (b)
3.          i++;
4.  }
```

Program

Compare Corresponding Statement Instances

```
1.  while (a){
2.      if (b)
3.          i++;
4.  }
1.  while (a){
2.      if (b)
3.
4.  }
```

Execution run π

```
1.  while (a){
2.      if (b)
3.          i++;
4.  }
1.  while (a){
2.      if (b)
3.          i++;
4.  }
```

Execution run π'

Compare Corresponding Statement Instances

```
1.  while (a){
2.      if (b)
3.          i++;
4.  }
1.  while (a){
2.      if (b)
3.
4.  }
```

Execution run π

```
1.  while (a){
2.      if (b)
3.          i++;
4.  }
1.  while (a){
2.      if (b)
3.          i++;
4.  }
```

Execution run π'

Fault Localization - branches

```
1.  while (a){
2.      if (b)
3.          i++;
4.  }
1.  while (a){
2.  if (b)
3.
4.  }
```

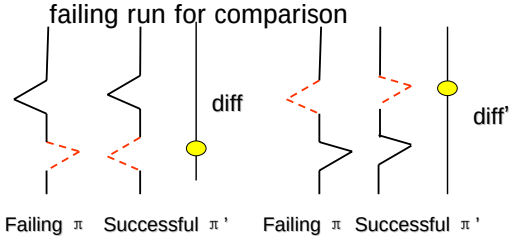
Execution run π

```
1.  while (a){
2.      if (b)
3.          i++;
4.  }
1.  while (a){
2.  if (b)
3.          i++;
4.  }
```

Execution run π'

Choose a Successful Run

- Comparison of Differences Select one failing run for comparison



Choose a Successful Run

- Comparison of Differences Select one failing run for comparison

Location of Branches

TCAS program

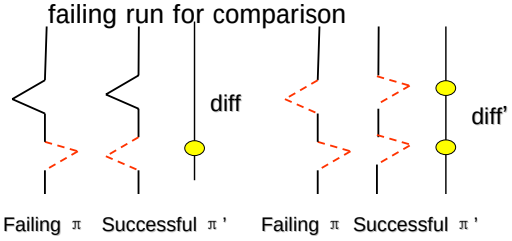
```
1.  int main(int argc, char **argv)
2.      if (argc < 3 ){
3.          printf("parameter error\n");
4.          return 0;
5.      }
6.
7.      if ( m == -1)
8.          ....
9.  }
```

check the input

Favor branches near to the observable error

Choose a Successful Run

- Comparison of Differences Select one failing run for comparison



Choose a Successful Run

- Comparison of Differences Select one failing run for comparison

Summary

- Computer-Aided Debugging
  - Automatically identify suspicious statements
- Future Trends
  - More accurate results
  - Novel way to use results of existing methods

Tools

- Testing
  - Numerous, Junit for unit testing
- Slicing
  - Several tools for static slicing
    - Codesurfer, Indus
  - Dynamic slicing – Jslice http://jslice.sourceforge.net
    - More naturally corresponds to debugging, than static slicing
- Fault Localization
  - Andreas Zeller's Askigor system

Assignment 2

- Debugging using Jslice tool
  - Dynamic Slicing Tool for Java
    - Demo of tool
    - Some description
    - Hands-on work on slicing/testing.

Slicing for Debugging

- Set criterion
  - Why is output v = 0 at end of program for input a = 2?
- Technique
  - Compute closure of dynamic data/control dep.
- Inspect the slice
  - Could possibly lead programmer to suspect other variables/line numbers
  - Slice again ….

Architecture of JSlice