

# *SOFTWARE DEBUGGING*

## *CS3213 FSE*

**Prof. Abhik Roychoudhury**

National University of Singapore



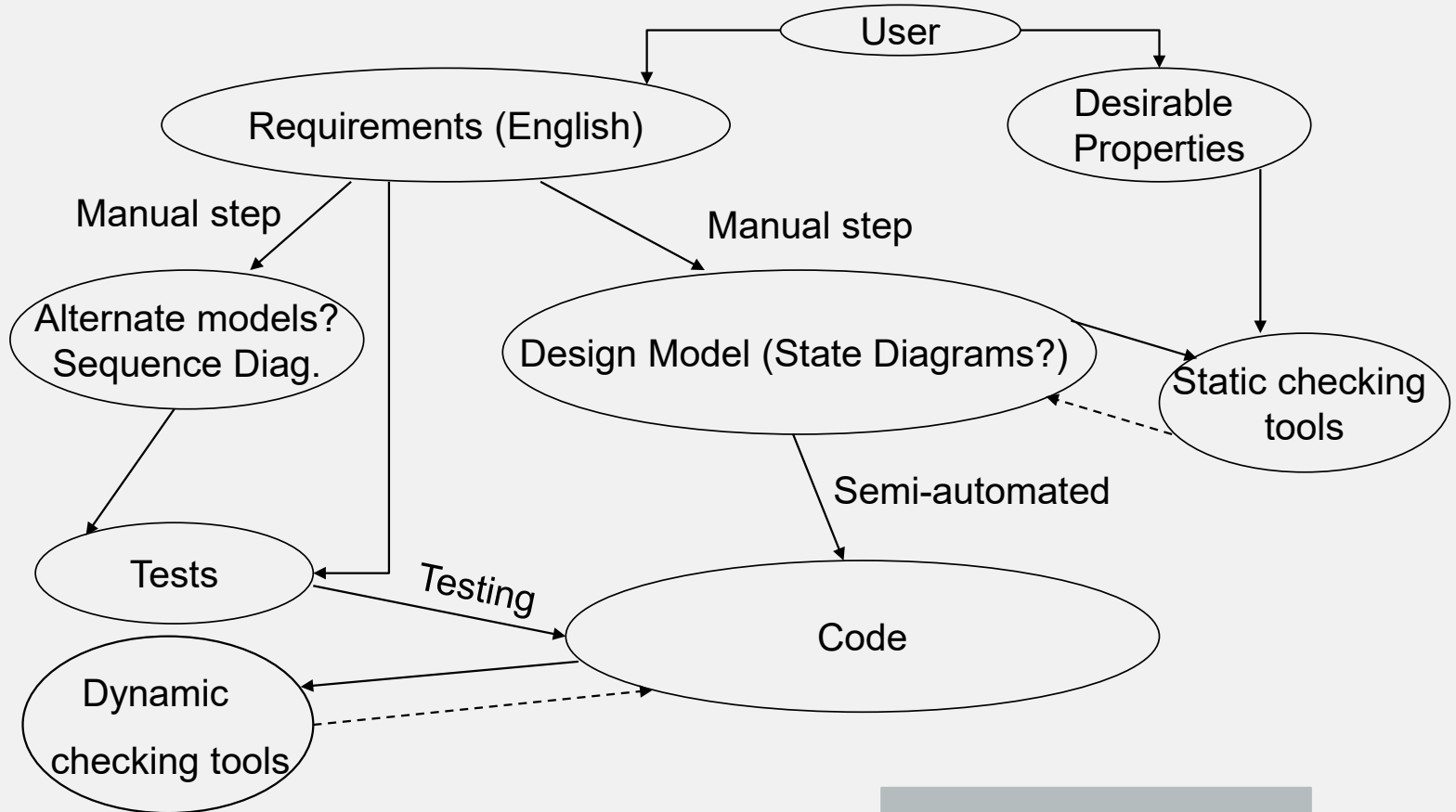
# WHAT WE DID EARLIER

- System Requirements: Use-cases, Scenarios, Sequence Diagrams
  - System structure: Class diagrams
  - Discussion on semantics
  - System behavior: State diagrams
  - Testing
- 
- Today
    - **Software Debugging**

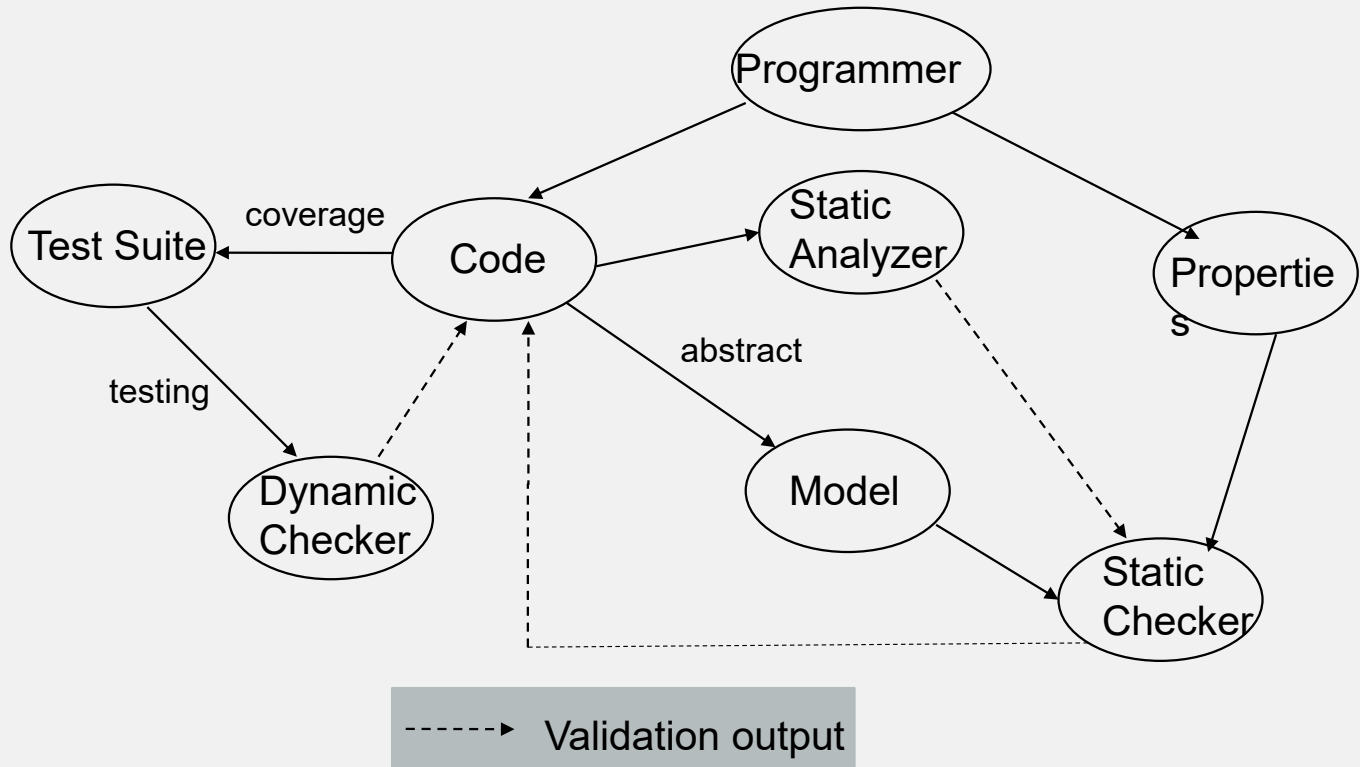
# SOFTWARE CONSTRUCTION

- From a design model
  - In safety-critical domains – automotive, avionics.
  - D0 I78C – software in airborne systems.
- Or, hand-constructed
  - Usual practice – audio, video and other domains.
  - UML models only for guidance.

# MODEL-DRIVEN ENGINEERING



NO MODEL MAY BE AVAILABLE.



# PROGRAMMING



Creativity

+



Precision

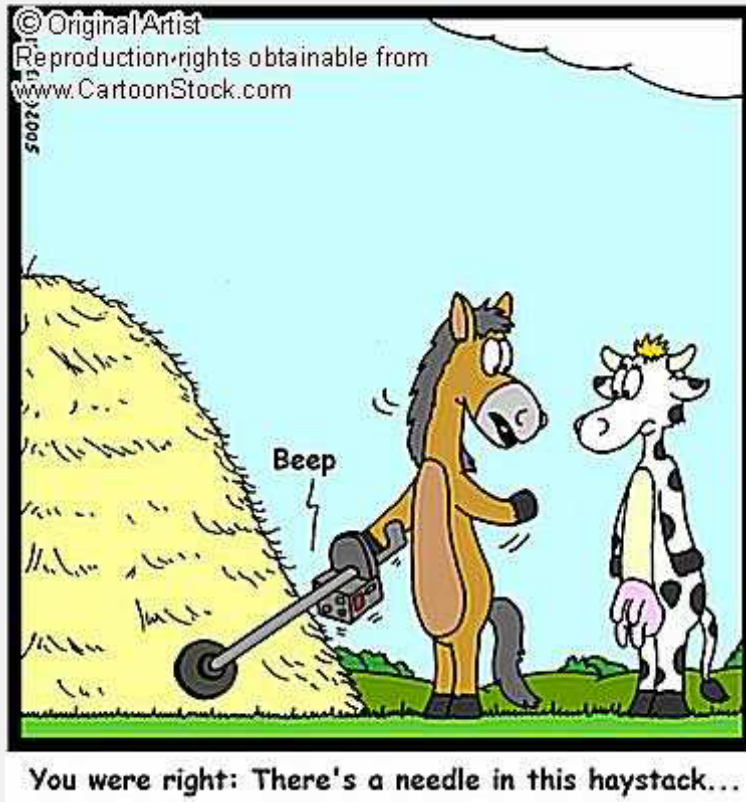
# THE **ART** OF DEBUGGING



"A **software bug** (or just "bug") is an error, flaw, mistake, ... in a computer program that prevents it from behaving as intended (e.g., producing an **incorrect result**). ... Reports detailing bugs in a program are commonly known as **bug reports**, fault reports, ... change requests, and so forth."

--- Wikipedia

# TOOLS?



We should **automatically** produce the bug report via analysis of program and/or execution trace

Bug report is a small **fragment** of the program.



# ORGANIZATION

- **Brief History of Debugging**
- Dynamic checking of programs
  - Dynamic slicing
  - Static vs. Dynamic slices
  - Fault Localization

## A QUOTE FROM 20 YEARS AGO

*“Even today, debugging remains very much of an art. Much of the computer science community has largely ignored the debugging problem..... over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools.”*

Hailpern & Santhanam, IBM Systems Journal, 41(1), 2002

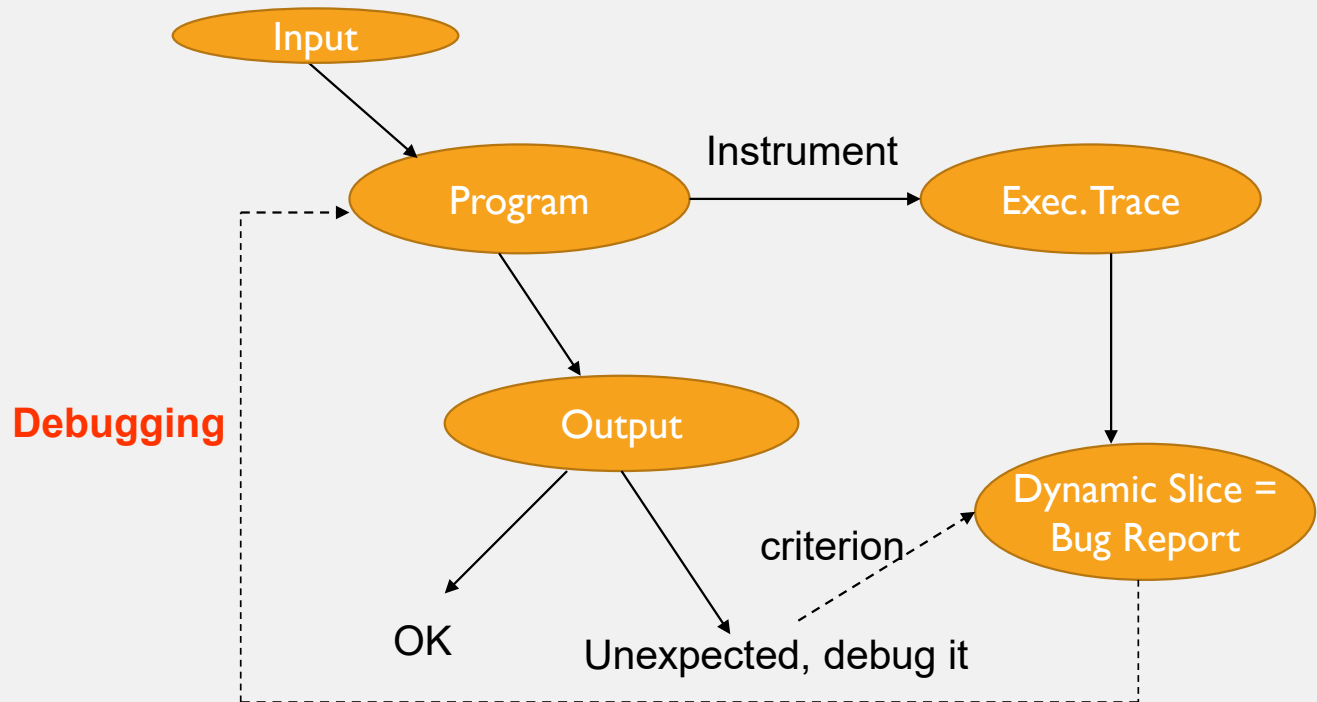
What about the current techniques, beyond breakpoints?

**Let us review them first.**

Any progress in 2002 – 2022?

**How can white-box analysis help? (we briefly discuss in week 9)**

# DYNAMIC SLICING: A 1<sup>ST</sup>- GENERATION DEBUGGING AID



# DYNAMIC SLICING

input a

Consider input a == 2

line 2 not included in slicing

Control  
Dependence

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

DV Pairs.

Data  
Dependence

Slicing  
Criterion

Based on  
unexpected value.

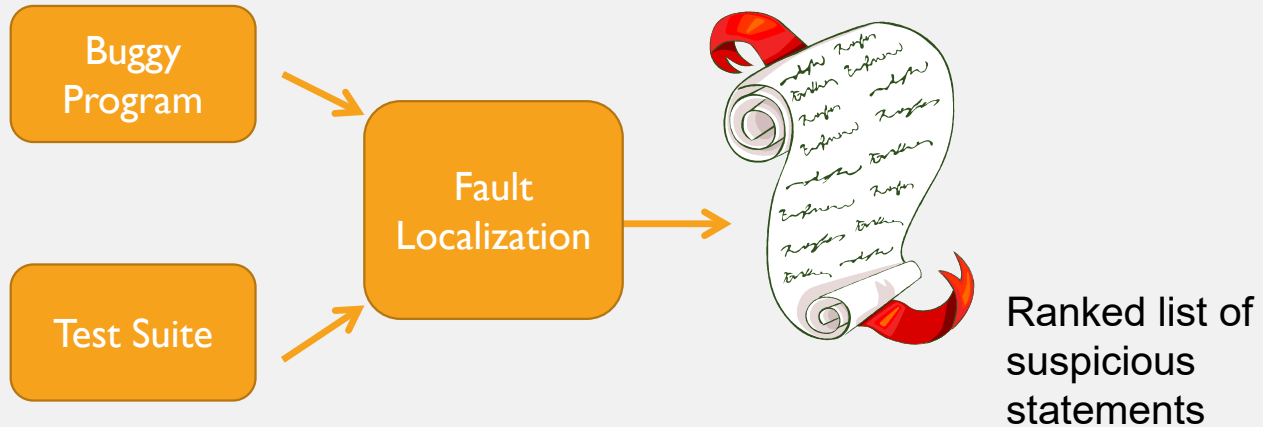
unexpected value in b.  
original value in x.

DV pair is (2,b), (5,b).  
Consider test a=2. the only DV pair is (5,b).

# DYNAMIC SLICING FOR DEBUGGING?

- Scalability
  - Large traces to analyze (and store?)
    - Optimizations exist – online compression.
  - Slice is too huge – slice comprehension
    - Tools such as WHYLINE have made it more user friendly
- Slicing still does not tell us what is actually wrong
  - Where did we veer off from the intended behavior?
  - What is the intended behavior? Often not documented! Lack of specifications is a problem.

# STATISTICAL FAULT LOCALIZATION



Assign scores to program statements based on their occurrence in passing / failing tests. **Correlation equals causation!**

*record statements that are executed.*

$$\text{Score}(s) = \frac{\frac{\text{fail}(s)}{\text{allfail}}}{\frac{\text{fail}(s)}{\text{allfail}} + \frac{\text{pass}(s)}{\text{allpass}}}$$

*passing tests where s evaluate at least once.*

*all failing tests*

An example of scoring scheme [Tarantula]

*Score = 1, when s perfectly correlates with failing test cases. → pass(s) = 0.*

# RANKED BUG REPORT

- We use the Tarantula toolkit.
- Given a test-suite  $T$

$$\text{Score}(s) = \frac{\frac{\text{fail}(s)}{\text{allfail}}}{\frac{\text{fail}(s)}{\text{allfail}} + \frac{\text{pass}(s)}{\text{allpass}}}$$

- $\text{fail}(s) \equiv \#$  of failing executions in which  $s$  occurs
- $\text{pass}(s) \equiv \#$  of passing executions in which  $s$  occurs
- $\text{allfail} \equiv \text{Total } \#$  of failing executions
- $\text{allpass} \equiv \text{Total } \#$  of passing executions
  - $\text{allfail} + \text{allpass} = |T|$
- Can also use other metric like Ochiai.

Name	Formula	Name	Formula
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$
Sørensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$	Dice	$\frac{2a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Kulczynski1	$\frac{a_{ef}}{a_{nf}+a_{ep}}$	Kulczynski2	$\frac{1}{2} \left( \frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ef}}{a_{ef}+a_{ep}} \right)$
Russell and Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Hamann	$\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$
Simple Matching	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Sokal	$\frac{2(a_{ef}+a_{np})}{2(a_{ef}+a_{np})+a_{nf}+a_{ep}}$
M1	$\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$	M2	$\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$
Rogers-Tanimoto	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{ep}+2(a_{nf}+a_{ep})}$	Goodman	$\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$
Hamming etc.	$a_{ef} + a_{np}$	Euclid	$\sqrt{a_{ef} + a_{np}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	Overlap	$\frac{a_{ef}}{\min(a_{ef}, a_{nf}, a_{ep})}$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}}}$	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep} + \frac{10000a_{ef}a_{ep}}{a_{ef}}}$
Ample	$\left  \frac{a_{ef}}{a_{ef}+a_{nf}} - \frac{a_{ep}}{a_{ep}+a_{np}} \right $	Wong1	$a_{ef}$
Wong2	$a_{ef} - a_{ep}$		
Wong3	$a_{ef} - h$ , where $h = \begin{cases} a_{ep} & \text{if } a_{ep} \leq 2 \\ 2 + 0.1(a_{ep} - 2) & \text{if } 2 < a_{ep} \leq 10 \\ 2.8 + 0.001(a_{ep} - 10) & \text{if } a_{ep} > 10 \end{cases}$		
Ochiai2	$\frac{a_{ef}a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Geometric Mean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Harmonic Mean	$\frac{(a_{ef}a_{np}-a_{nf}a_{ep})((a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np}))}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Arithmetic Mean	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Cohen	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{ep})+(a_{ef}+a_{nf})(a_{nf}+a_{np})}$		
Scott	$\frac{4a_{ef}a_{ep}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
Fleiss	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
Rogot1	$\frac{1}{2} \left( \frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}} + \frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}} \right)$		
Rogot2	$\frac{1}{4} \left( \frac{a_{ef}}{a_{ef}+a_{ep}} + \frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{np}}{a_{np}+a_{ep}} + \frac{a_{np}}{a_{np}+a_{nf}} \right)$		

Can use several other available metrics for ranking statements, e.g. Ochiai metric

$$\text{Score(s)} = \frac{\text{fail(s)}}{\sqrt{\text{allfail} * (\text{fail(s)} + \text{pass(s)})}}$$

A model for spectra-based software diagnosis, Naish et. al., TOSEM 20(3), 2011.



# PASSING AND FAILING TESTS ON THE BUGGY VERSION

```

1  input a, b, c;
2  lo, mid, hi = a, b, c;
3  if (mid < lo)
4    lo, mid, hi = mid, lo, hi
5  if (hi < lo)
6    lo, mid, hi = hi, lo, mid
7  if (hi < mid)
8    lo, mid, hi = hi, lo, mid
9  print lo, mid, hi
10 assert lo < mid < hi
    
```

Sort on 3 distinct numbers

1,2,3	0,3,1	2,3,1
v	v	v
v	v	
x	x	
v	v	
x	x	
v	v	
x	v	
v	v	
succes	failure	succes
s		s

Choice of which  
passing test to  
compare matters

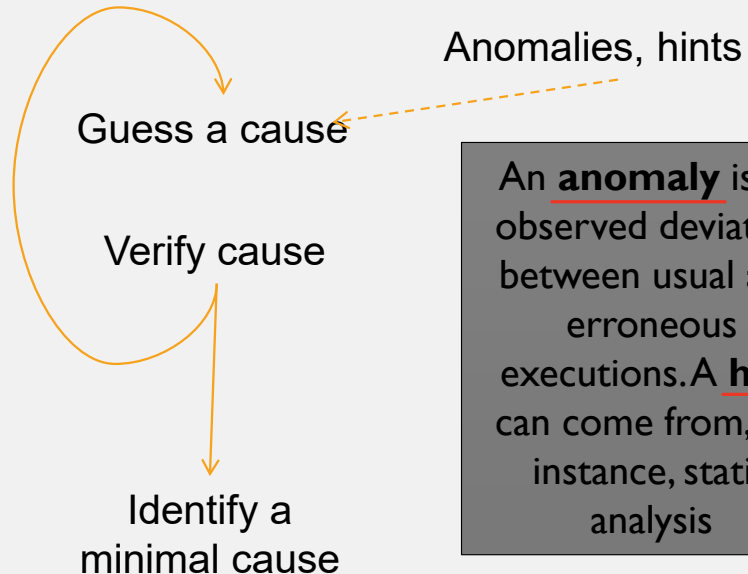
Is failure  
correlated with  
execution of  
statement 8 ?

Is failure correlated  
with execution of 8?  
*Non* execution of  
6?

# FAILURES AND CAUSES

A failure is an effect of some cause: elimination/workaround of the cause should remove the effect.

A cause could be some part of the input, some fragment of the code, or some part of the environment

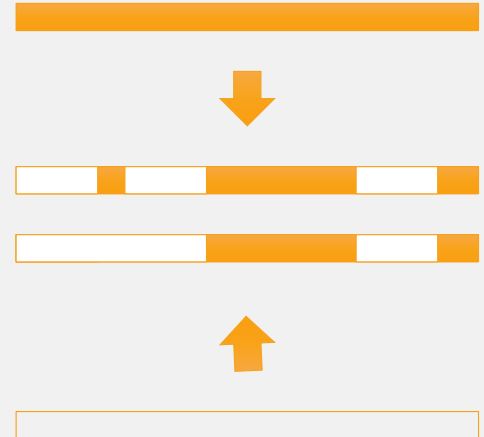


An anomaly is an observed deviation between usual and erroneous executions. A hint can come from, for instance, static analysis

A. Zeller: Why Programs Fail, A Guide to Systematic Debugging

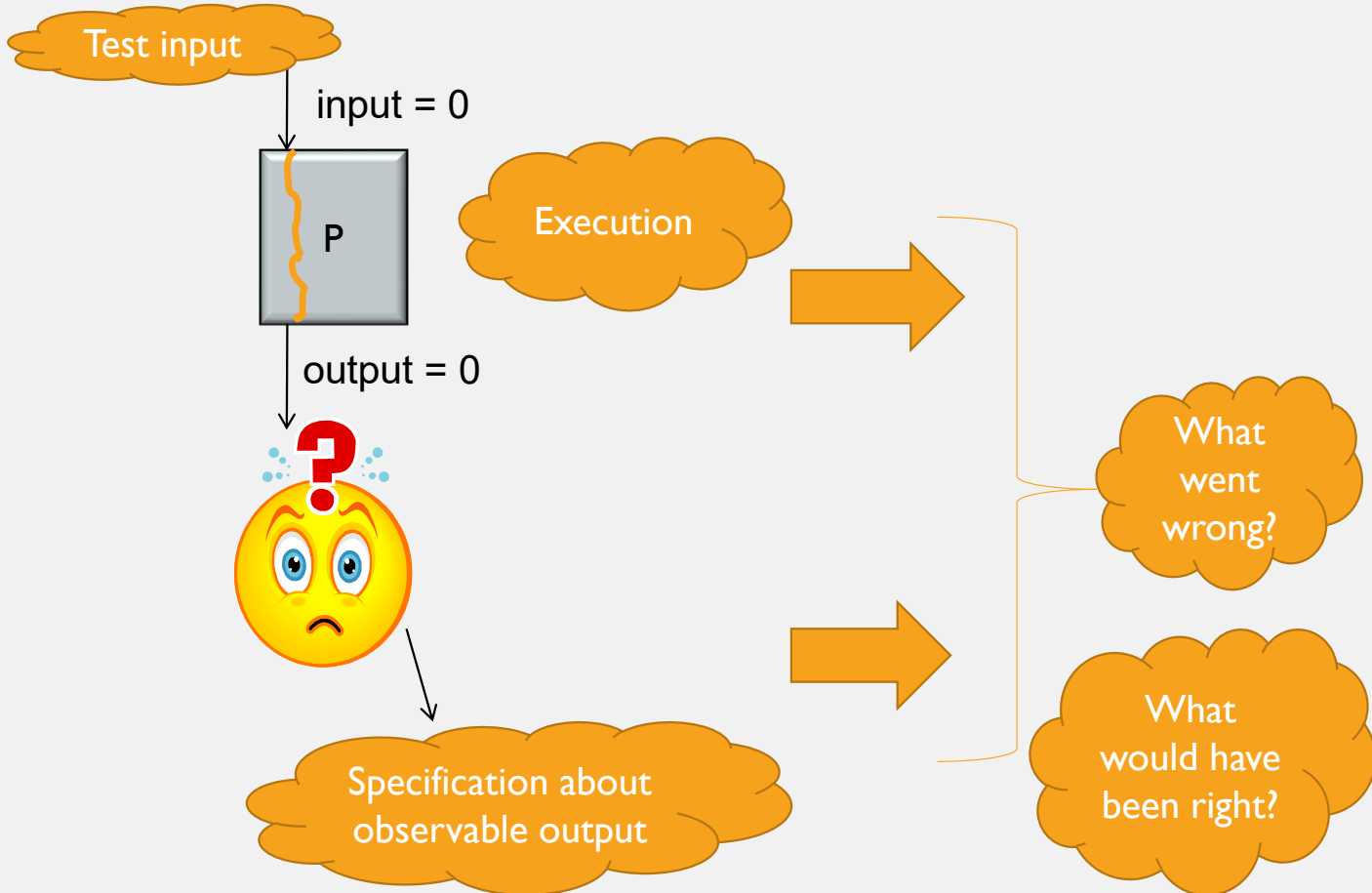
# ISOLATING FAILURE CAUSES A LA DELTA DEBUGGING

- How to figure out a minimal cause that ‘explains’ an error?
- Use a variation on binary search: narrow the difference between passing and failing inputs
  - Can do it on code (old version to new version)
  - On thread schedules

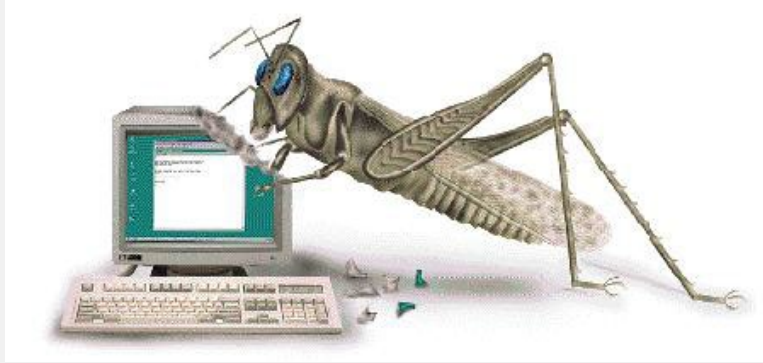


A. Zeller: Why Programs Fail, A Guide to Systematic Debugging

# THE ROLE OF SPECIFICATIONS



# SW DEBUGGING: SOCIAL ASPECTS



Software-controlled devices are ubiquitous ---  
automotive control, avionics control and consumer electronics  
Many of these software are **safety-critical**  
⇒ should be validated extensively.

# SW DEBUGGING: ECONOMICS

- How often do bugs appear ?
- How many of them are critical?
- How much money does a company gain by using sophisticated debugging tools?
- Could it be avoided simply by sparing one more programmer?

# SW DEBUGGING: ECONOMICS

- SW project with **5 million** LOC (note: Windows Vista is 50 million LOC !!)
  - Assume **linear scaling up** of errors
    - Actually could be more errors --- we make more mistakes as the SW grows long and arduous.
  - **1 hr to** fix each major error
    - Actually much more
  - **\$50K** salary per year

$$13 * \frac{5000000}{1000} = 65,000 \text{ bugs}$$

$$\frac{65,000}{44} \text{ weeks} = 1477 \text{ weeks} = \frac{1477}{50} \approx 30 \text{ years} = \$1.5 \text{ M}$$

## SW DEBUGGING: TOOLS

*“Even today, debugging remains very much of an art. Much of the computer science community has largely ignored the debugging problem..... over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools.” (Hailpern & Santhanam, IBM Sys Jnl, 41(1), 2002)*

- > Need methods and tools to trace back to the root cause of bug from the manifested error
- > What about the current tools?



# TYPICAL TOOLS

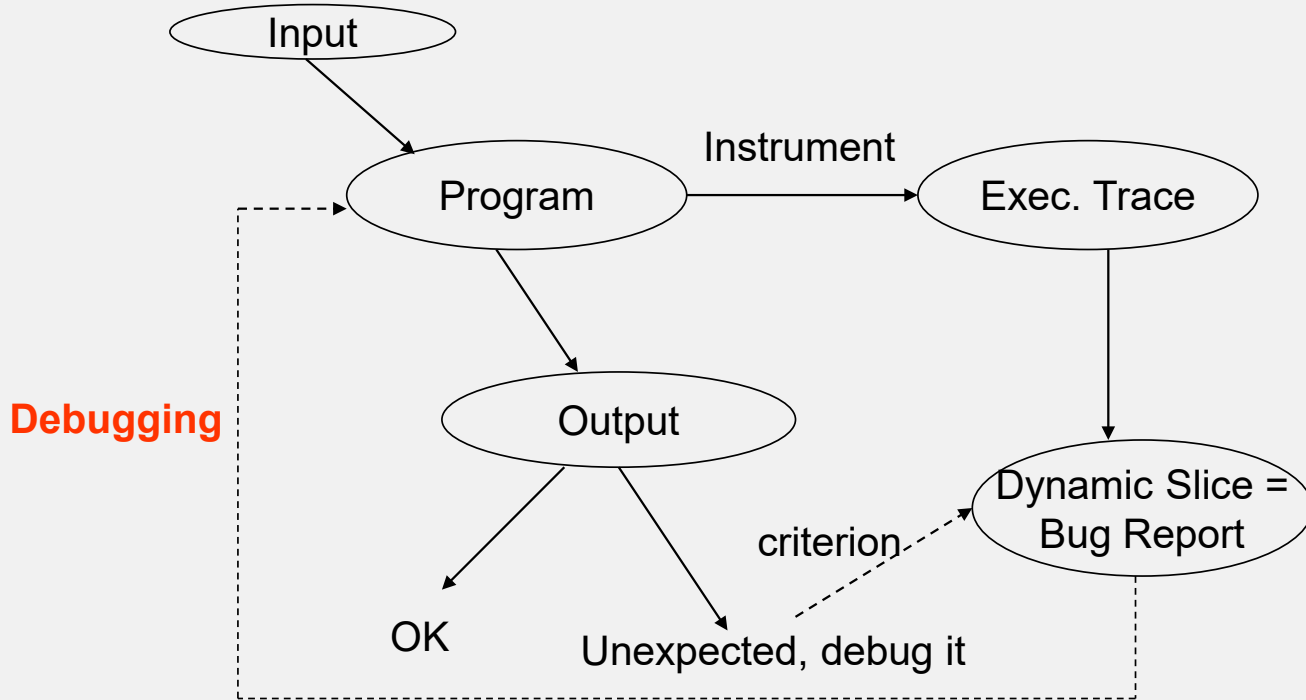
## Typical tools

- User can set breakpoints, and
- Replay an execution, and
- Watch it at the breakpoints.
- Lack of GUI is **not** the issue here.
  - Can easily collect and visualize more program info.
- **Lack of automation is the problem!**
  - **Need automated trace analysis.**

# ORGANIZATION

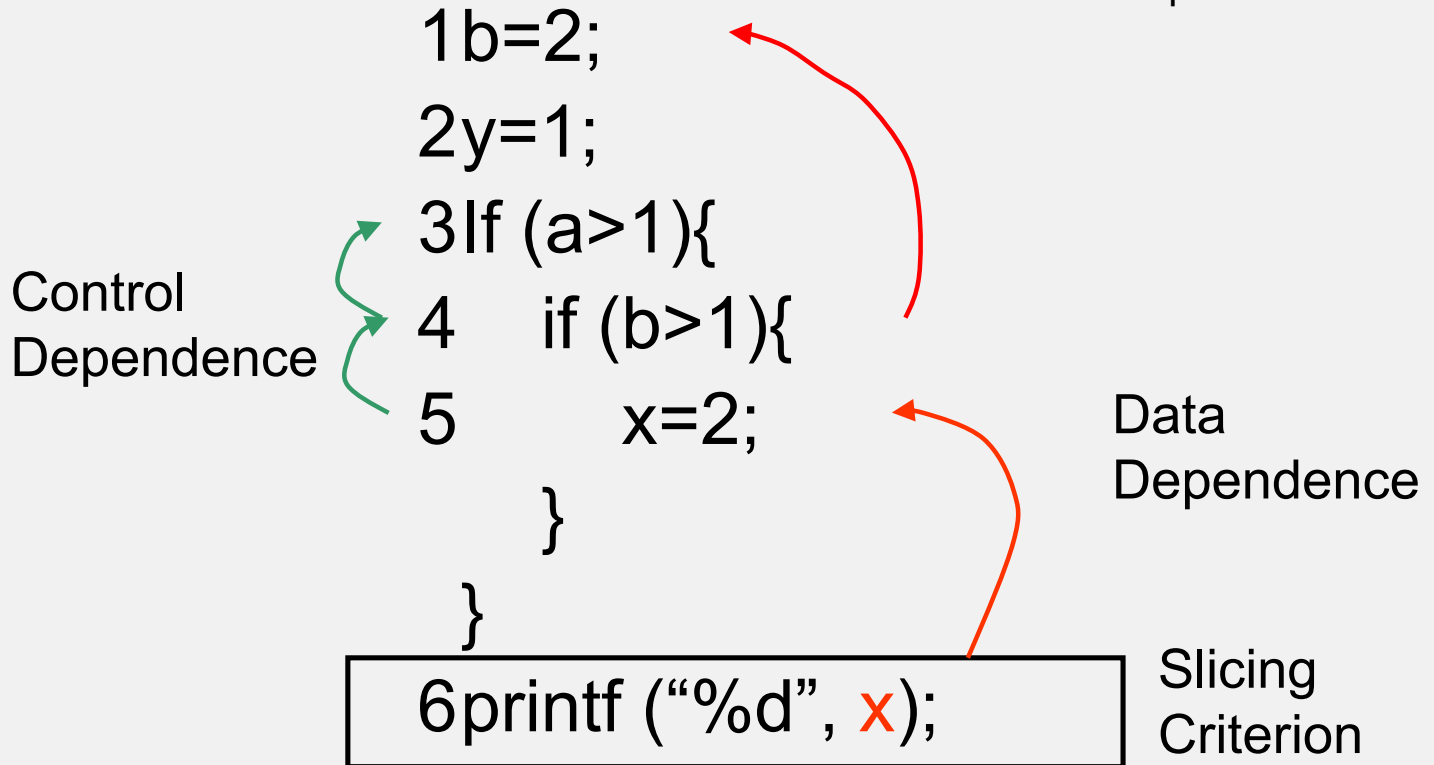
- Brief History of Debugging
- **Dynamic checking** of programs
  - Dynamic slicing
  - Hierarchical slicing
  - Fault Localization
  - Directed testing (next week)

# DYNAMIC SLICING FOR DEBUGGING



# DYNAMIC SLICING

Consider input  $a == 2$

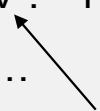


# DYNAMIC SLICE

- Set slicing criterion
  - (Variable  $v$  at first instance of line 70)
  - The value of variable  $v$  at first instance of line 70 is unexpected.
- Dynamic slice
  - Closure of
    - Data dependencies &
    - Control dependencies
  - from the slicing criterion.

# DYNAMIC DATA DEPENDENCIES

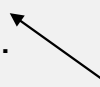
V := 1;  
...  
U := V



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

Not an issue. At runtime, we alr know if  $i = j$ .

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



→ Do we consider this data dependence edge ?

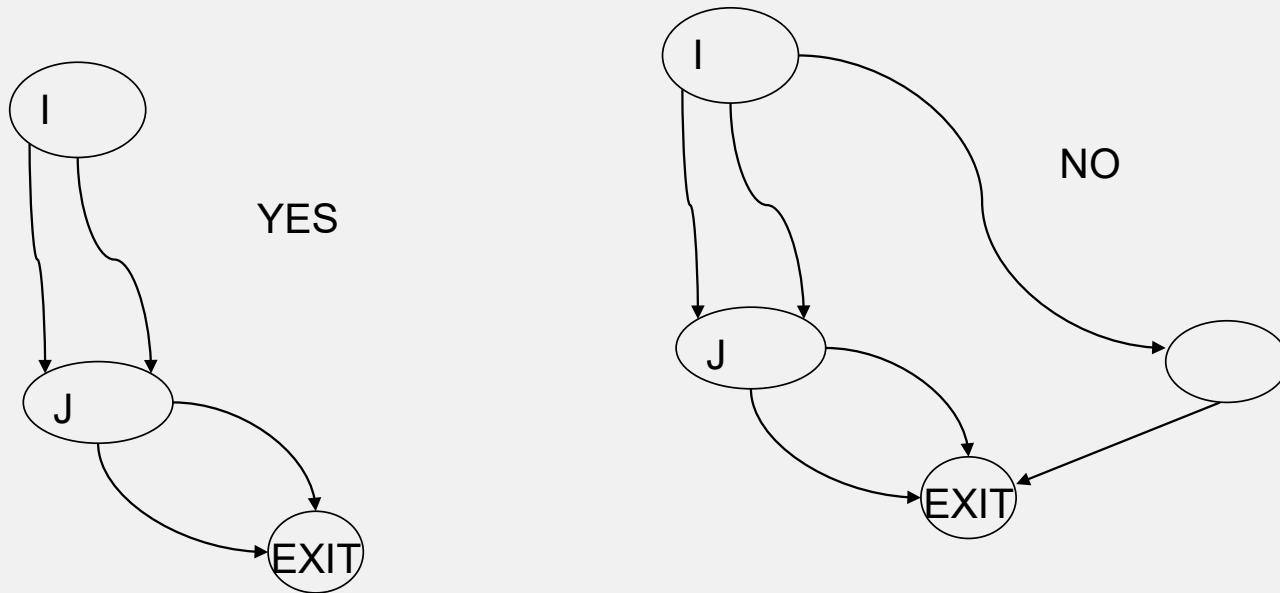
→ Remember that the slicing is for an input, so the addresses are resolved

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

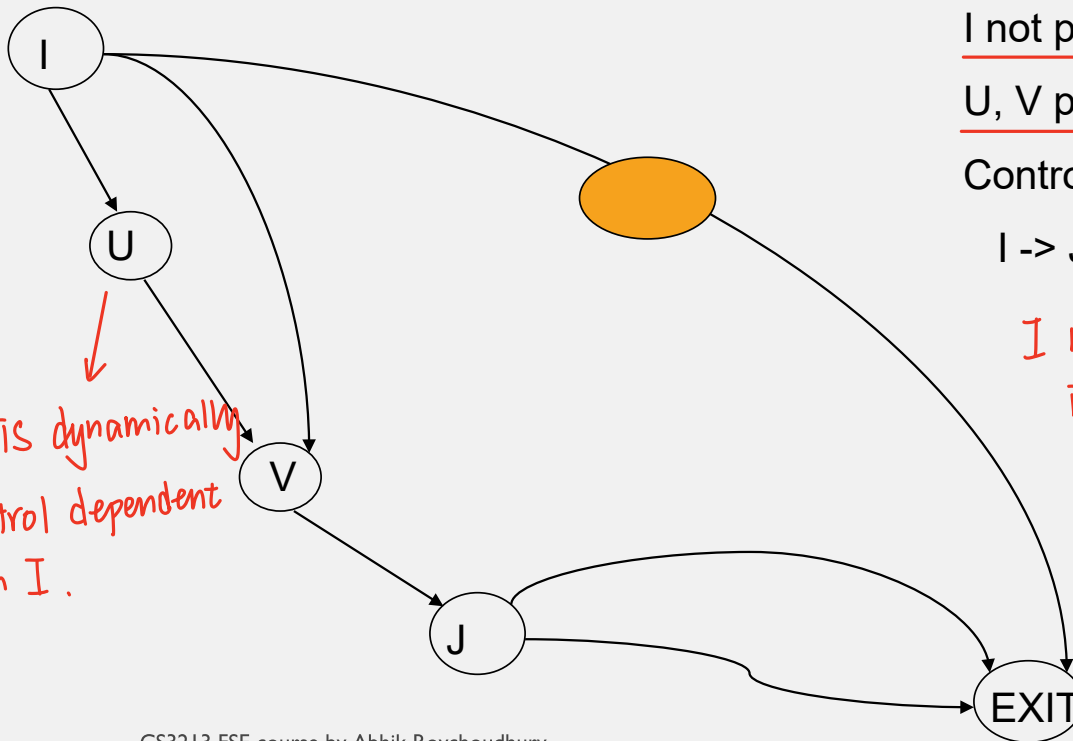
# STATIC CONTROL DEPENDENCIES

**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



# STATIC CONTROL DEPENDENCIES



I not post-dom by J

U, V post-dom by J

Control dependence

$I \rightarrow J$

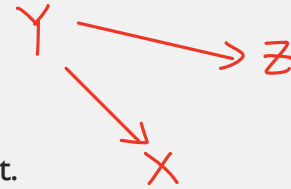
*I makes decision whether J is executed or not.*

*U is dynamically control dependent on I.*



# DYNAMIC CONTROL DEPENDENCIES

- X is dynamically control dependent on Y if
  - Y occurs before X in the execution trace
  - X's stmt. is statically control dependent on Y's stmt.
  - No statement Z between Y and X is such that X's stmt. is statically control dependent on Z's stmt.
- Captures the intuition:
  - What is the nearest conditional branch statement that allows X to be executed, in the execution trace under consideration.



# 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

# ORGANIZATION

- Dynamic checking of programs
  - Dynamic slicing
  - Static vs dynamic slices
  - Fault Localization
  - Directed testing

# STATIC VS DYNAMIC SLICING

- Static Slicing → All dependencies.
  - source code
  - statement
  - static dependence
- Dynamic Slicing → Only stmts. evaluated given different inputs.
  - a particular execution
  - statement instance
  - dynamic dependence

## STATIC VS DYNAMIC SLICING

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


static slicing.

(dynamic either one only).

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?

① at run-time, p, q  $\uparrow$  maybe,

② compile-time,  $3 \rightarrow 2$

- Static points-to analysis is always conservative

# RELEVANT SLICING

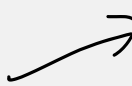
In between static and dynamic

Relevant :  
Dynamic + potential.

dynamic:  
1 b=10;  
2 x=1; only consider influence of statements  
3 If (a>1){ that are exists in  
4 if (b>1){ the faulty execution trace  
5 x=2; X stmt that aren't  
6 } executed.  
7 }  
8 }

6 printf ("%d", x);

## RELEVANT SLICING

source of error. 

```
1 b=1;  
2 x=1;  
3 if (a>1){  
4     if (b>1){  
5         x=2;  
6     }  
7 }  
8 printf ("%d", x);
```



① 5 is not executed:

b data depends on 2. 2 depends on nothing.  
So, only 2, b in dynamic slicing.

## RELEVANT SLICING

② If 5 executed:

b data depends on 5; 5 control depend on 4;  
4 control depend on 3; 4 data depend on 1;

input: a=2

Dynamic trace : 1, 2, 3, 4, b.

Source of Failure

Dynamic Slice

Execution is omitted

1 b=1;

2 x=1;

3 If (a>1){

4 if (b>1){

5

x=2;

}

}

6 printf ("%d", x);

Incorrect omission.

# POTENTIAL DEPENDENCE

input: a=2

1 b=1;

2 x=1;

3 if (a>1){

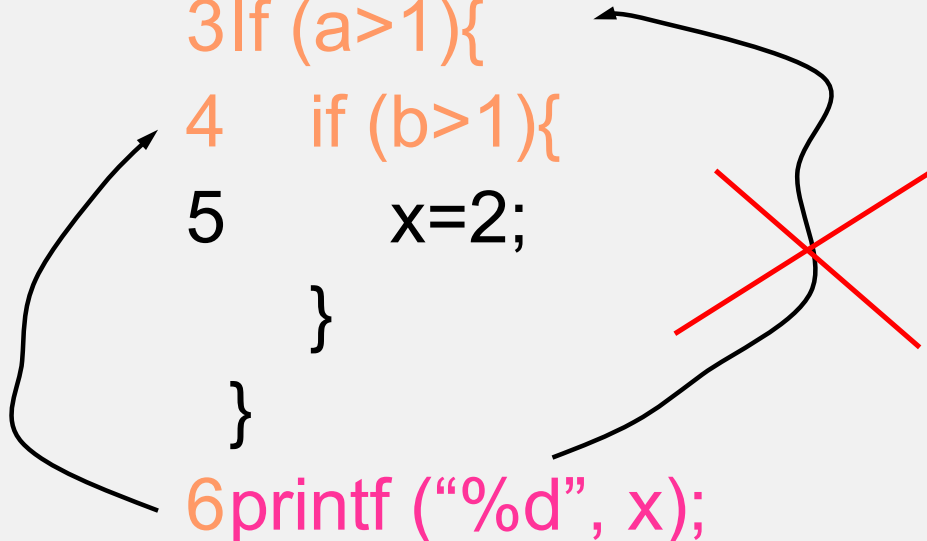
4     if (b>1){

5         x=2;

       }

   }

6 printf ("%d", x);



# RELEVANT SLICE

Consider branch, if evaluated differently, would have then evaluated an omitted stmt, that is data dependent with slicing criterion

input: a=2

Potential  
Dependence

If line 4 is executed correctly, it would then execute some stmt that affect the data dependency

```
1 b=1;
2 x=1;
3 if (a>1){
4   if (b>1){
5     x=2;
6   }
7 }
8 printf ("%d", x);
```

Dynamic Data  
Dependence

# PROGRAM SLICE

Static      Dynamic      Relevant

1

1

1 b=1;      input: a=2

2

2

2

2 x=1;

3

3 If (a>1){

4

↑

4

4 if (b>1){      DV-pair between (1,4).

5

∴ 5 is not executed.

5

x=2;

∴ b depends on 2.  
2 depends on nothing.

}

}

6

6

6

6 printf ("%d", x);

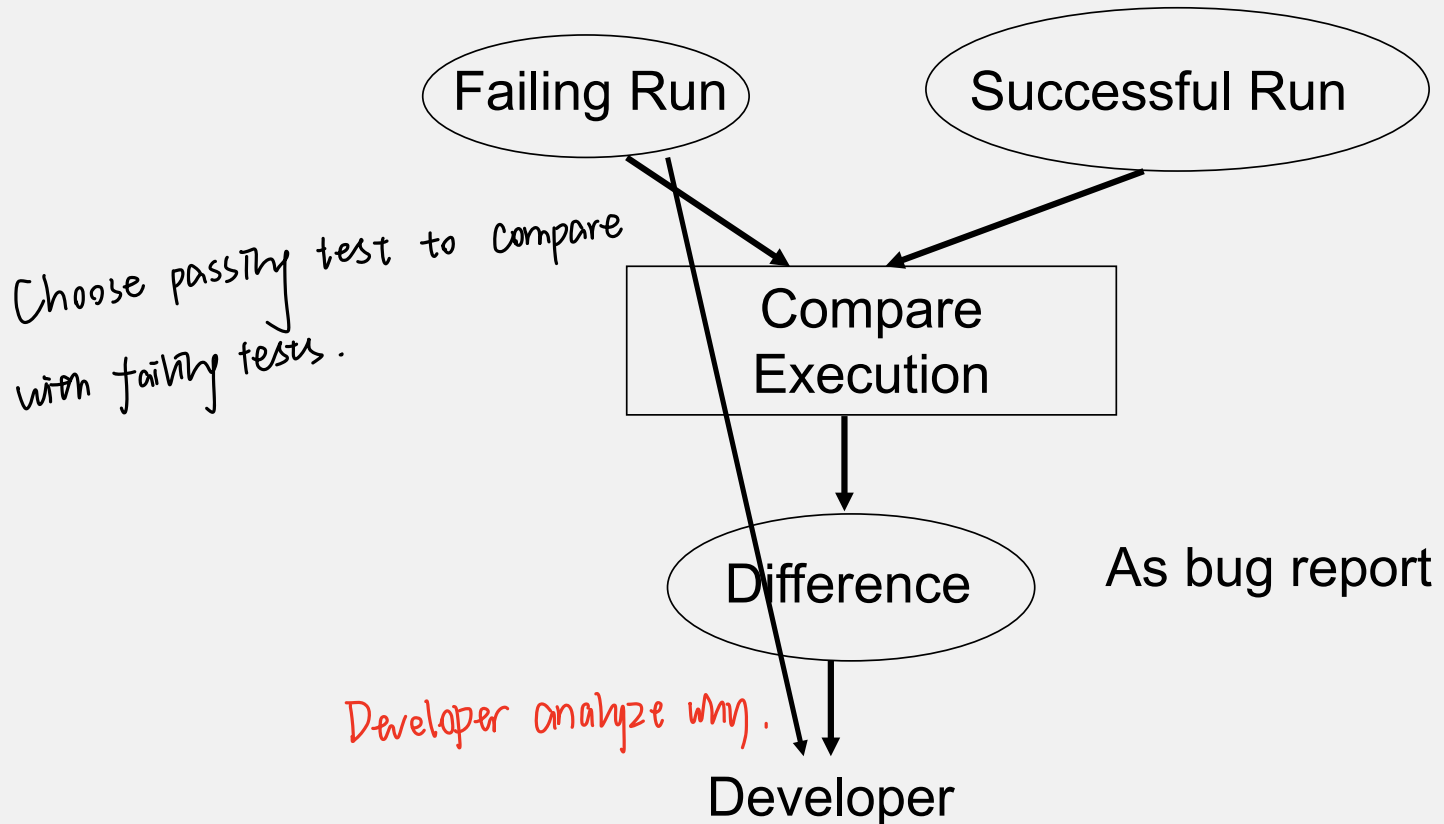
# ORGANIZATION

- Dynamic checking of programs
  - Dynamic slicing
  - Hierarchical slicing
  - Fault Localization

## MORE ON DEBUGGING

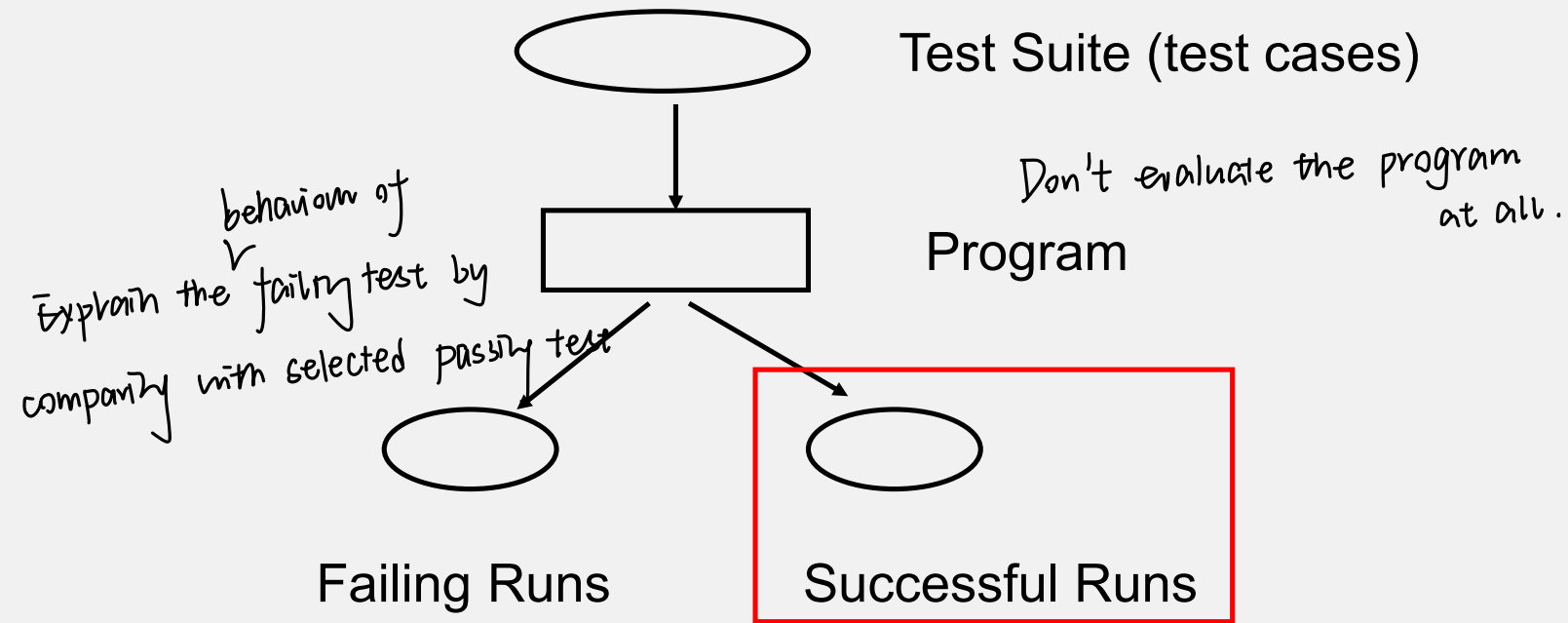
- Dynamic slicing analyzes the problematic execution trace.
  - Problematic: output is unexpected
  - OK: output is as expected.
- Alternatively:
  - We could compare a given problematic trace with a passing trace to localize the source of error.

# FAULT LOCALIZATION: OVERVIEW



# TESTING BASED FAULT LOCALIZATION

- Use testing to help debugging

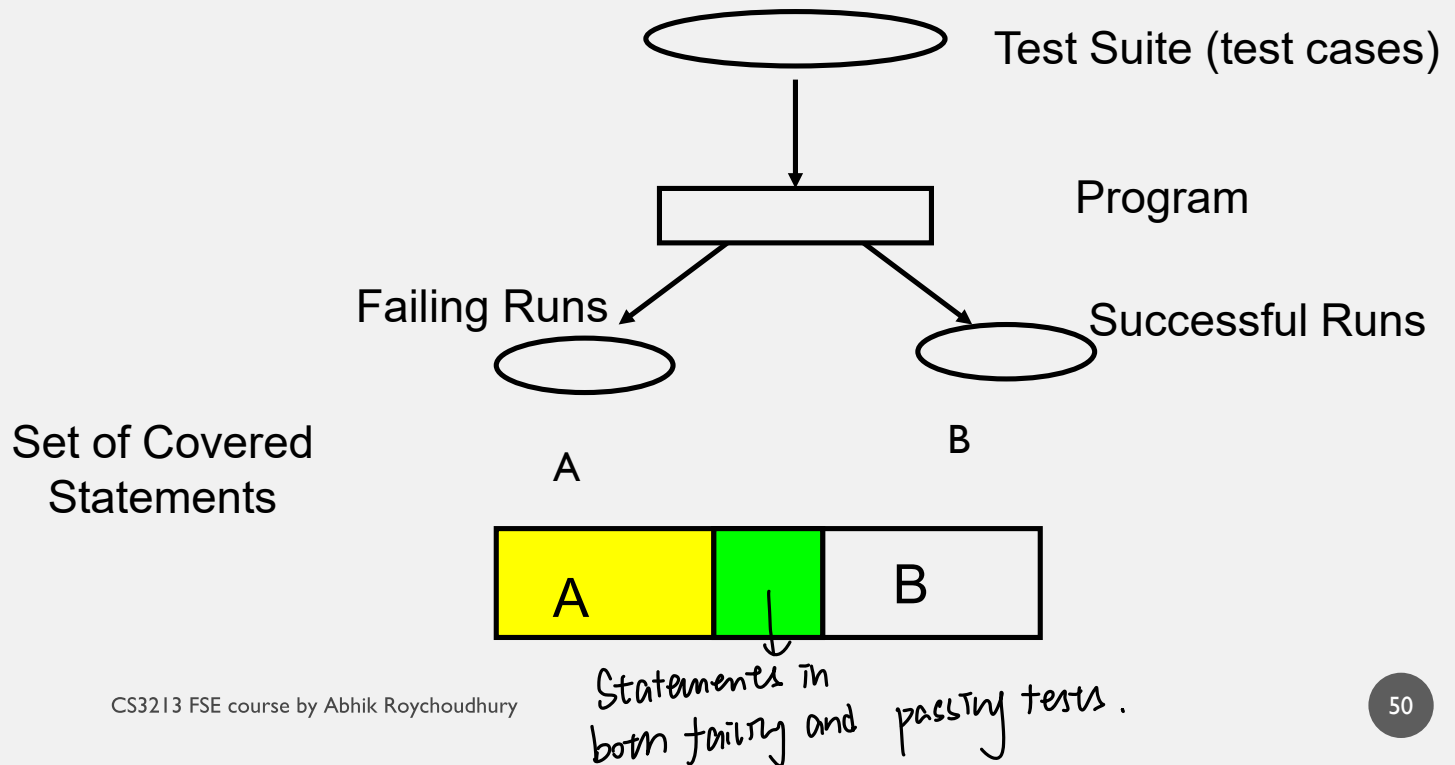




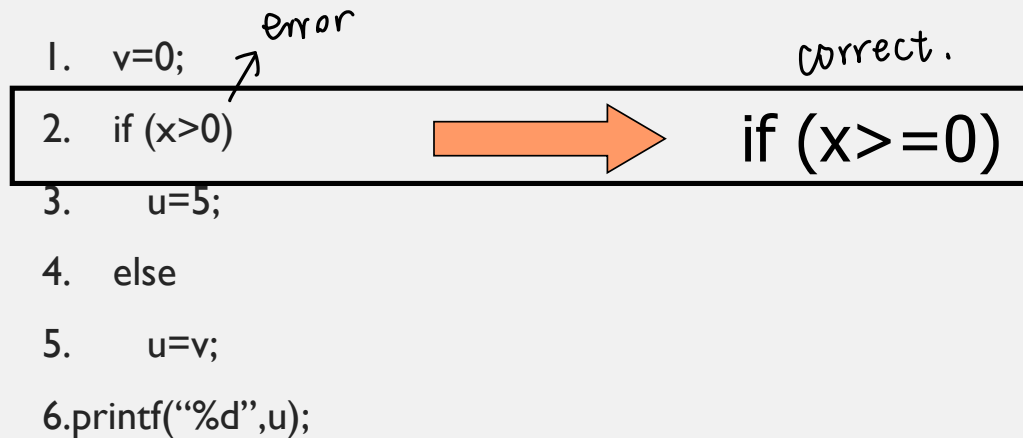
# TESTING BASED FAULT LOCALIZATION

- What to Compare
  - choice of the Execution Run
- How to Compare
  - statement / basic block
  - predicates / branch statement
  - potential invariants
  - variable values

# FAULT LOCALIZATION - STATEMENT



## FAULT LOCALIZATION - BRANCHES



## FAULT LOCALIZATION - BRANCHES

1. `v=0;` To highlight branch differences.

2. `if (x>0)`

4. `else`

5. `u=v;`

6. `printf("%d",u);`

*Failing run, x=0*

1. `v=0;`

2. `if (x>0)`

3. `u=5;`

6. `printf("%d",u);`

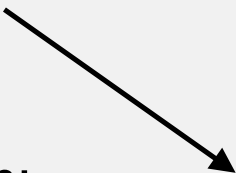
*Successful run, x=1*

# COMPARING EXECUTIONS

```
1. m=...
2. if (m >= 0) {
3.     ...
4.     lastm = m;
5. }
6. ....
```

should be

if ((m >= 0) && (lastm!=m))



## COMPARING EXECUTIONS

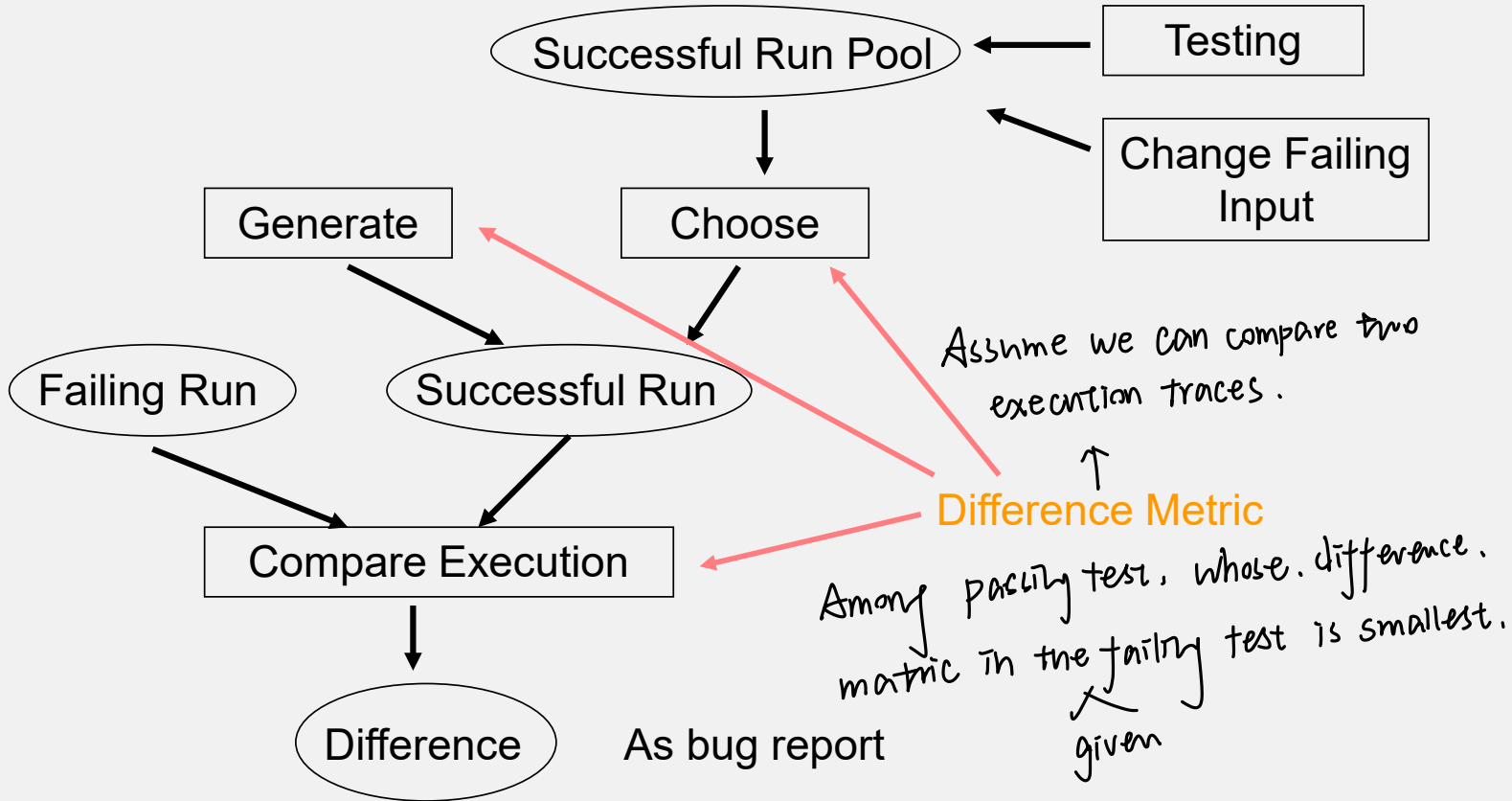
```
1.  m=...
2.  if (m >= 0) {
3.      ...
4.      lastm = m;
5.  }
6.  .....
```

*Failing run*

```
1.  m=...
2.  if (m >= 0) {
3.      ...
4.      lastm = m;
5.  }
6.  .....
```

*Successful run*

# FAULT LOCALIZATION



## EXAMPLE PROGRAM

*Program*

```
1.  if (a)
2.      i = i + 1;
3.  if (b)
4.      j = j + 1;
5.  if (c)
6.      if (d)
7.          k = k + 1;
8.      else
9.          k = k + 2;
10. printf("%d", k);
```



# COMPARING EXECUTIONS

1. if (a)

2.  $i = i + 1;$

3. if (b)

4.  $j = j + 1;$

5. if (c)

6. if (d)

7.  $k = k + 1;$

8. else

9.  $k = k + 2;$

10. printf("%d", k);

Execution run  $\pi$  *fail*

CS3213 FSE course by Abhik Roychoudhury

1. if (a)

2.  $i = i + 1;$

3. if (b)

4.  $j = j + 1;$

5. if (c)

6. if (d)

7.  $k = k + 1;$

8. else

9.  $k = k + 2;$

10. printf("%d", k);

Execution run  $\pi_1$  *pass*

Compare  $\pi$  with  $\pi_1, \pi_2, \dots \rightarrow$  find out where  $\pi$  and  $\pi_n$  has least branch diff.

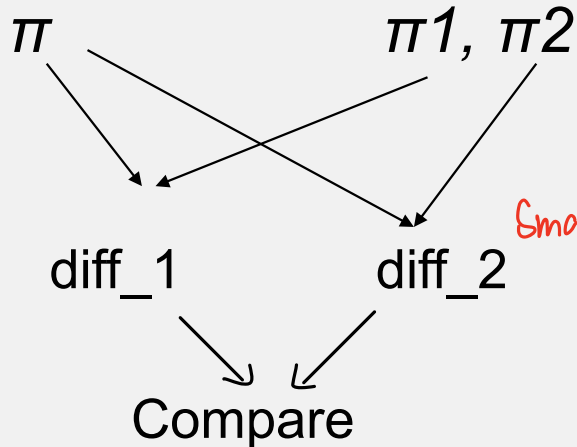
## SET OF STATEMENTS

- $S$  = Set of statements executed in  $\pi$ 
  - $\{1,3,5,6,7,10\}$
- $SI$  = Set of statements executed in  $\pi I$ 
  - $\{1,3,4,5,6,9,10\}$
- If  $\pi$  is faulty and  $\pi I$  is passing
  - Bug report =  $S - SI = \{4,7\}$
- Choice of the execution run to compare with is very important.

# ANOTHER DIFFERENCE METRIC

Failing Run

Successful Runs



- Number of Branches

- Location of Branches

Smaller difference: ① less # of different branches.  
②. difference occurs later

$diff\_1$  and  $diff\_2 \rightarrow$  find run with least diff

## DIFFERENCE B/W TRACES SHOWN

```
1.  if (a)
2.      i = i + 1;
3.  if (b)
4.      j = j + 1;
5.  if (c)
6.      if (d)
7.          k = k + 1;
8.      else
9.          k = k + 2;
10. printf(“%d”, k);
```

```
1.  if (a)
2.      i = i + 1;
3.  if (b)
4.      j = j + 1;
5.  if (c)
6.      if (d)
7.          k = k + 1;
8.      else
9.          k = k + 2;
10. printf(“%d”, k);
```

# TRACE ALIGNMENT AND DIFFERENCES

joining



Execution Run			Alignment				Difference
$\pi$	$\pi'$	$\pi''$	$\pi$	$\pi'$	$\pi'$	$\pi''$	$diff(\pi, \pi') \prec diff(\pi', \pi'')$
$1^1$	$1^1$	$1^1$					
$2^2$	$2^2$	$2^2$					
$3^3$	$3^3$	$3^3$					
$4^4$		$4^4$					
$5^5$		$5^5$					
$7^6$	$7^4$	$7^6$					
$8^7$	$8^5$						
$9^8$	$9^6$						
		$12^7$					
$1^9$	$1^7$	$1^8$					
$2^{10}$	$2^8$	$2^9$					
$3^{11}$	$3^9$	$3^{10}$					
$4^{12}$	$4^{10}$	$4^{11}$					
$5^{13}$	$5^{11}$	$5^{12}$					
$7^{14}$	$7^{12}$	$7^{13}$					
$8^{15}$							
$9^{16}$							
	$12^{13}$	$12^{14}$					
$14^{17}$	$14^{14}$	$14^{15}$					

# COMPARE CORRESPONDING STATEMENT INSTANCES

1. while (a){ *Align.*

2. if (b)

3. i++;

4. }

1. while (a){

2. if (b)

3.

4. }

1. while (a){

5. ....

1. while (a){

2. if (b)

3. i++;

4. }

1. while (a){

2. if (b)

3. i++;

4. }

1. while (a){

2. if (b)

1<sup>st</sup> Loop  
Iteration

2<sup>nd</sup> Loop  
Iteration

3<sup>rd</sup> Loop  
Iteration

Use control dependencies!

↓  
*not aligned.*

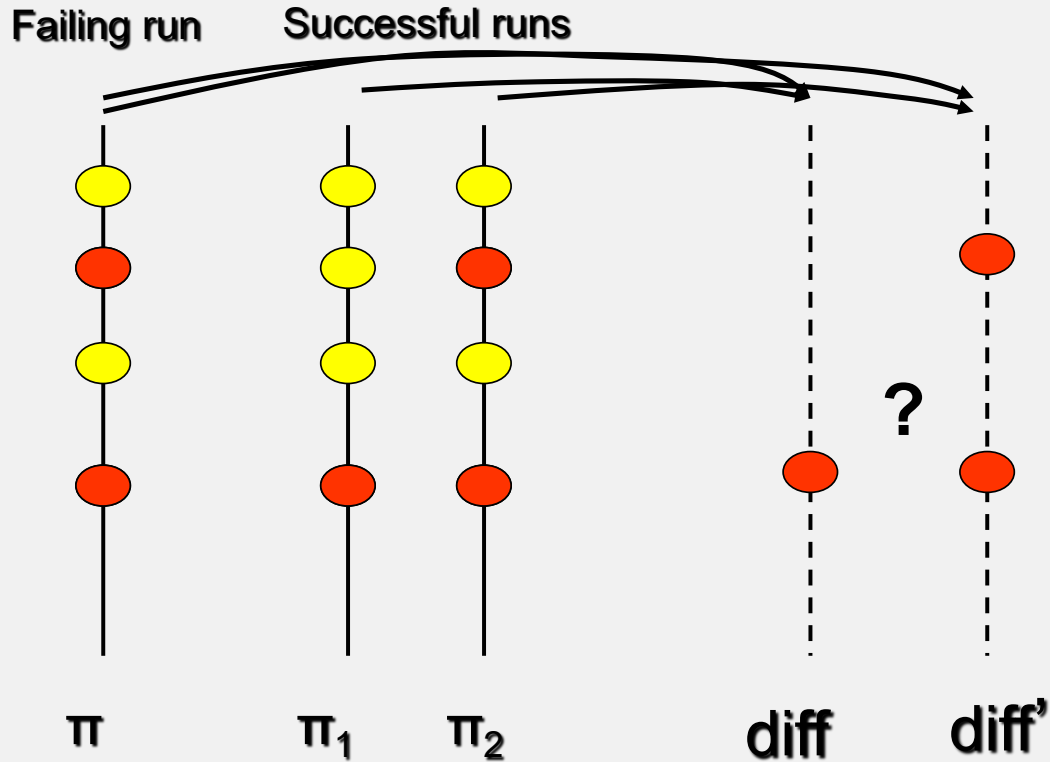
# FORMAL NOTION OF ALIGNMENT

*Align traces using control dependencies.*

For any pair of event  $e$  in run  $x$  and event  $e_0$  in run  $y$ , we define  $\text{align}(e, e_0) = \text{true}$  ( $e$  and  $e_0$  are aligned) iff.

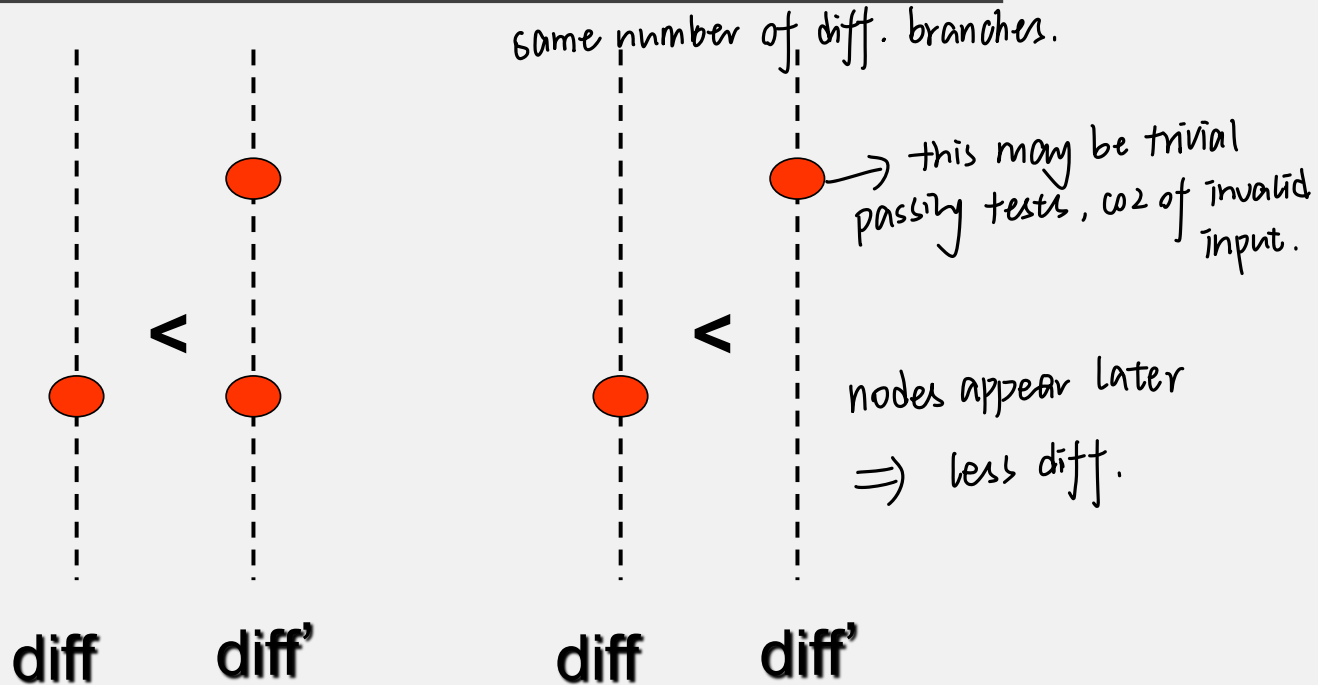
- $\text{stmt}(e) = \text{stmt}(e_0)$ , and
- either
  - $e, e_0$  are the first events appearing in runs  $x, y$  or
  - $\text{align}(\text{dep}(e, x), \text{dep}(e_0, y)) = \text{true}$ .
  - $\text{dep}(e, x) ==$  the event on which  $e$  is dynamically control dependent in run  $x$ .

# COMPARISON OF DIFFERENCES





# COMPARISON OF DIFFERENCES



# LOCATION OF BRANCHES IS IMPORTANT

tests that take Invalid input are trivial passing tests.

```
1. int main(int argc, char **argv)
```

```
2.     if (argc < 3 ){
```

```
3.         printf("parameter error\n");
```

```
4.         return 0;
```

```
5.     }
```

*check the input*

```
6.     ....
```

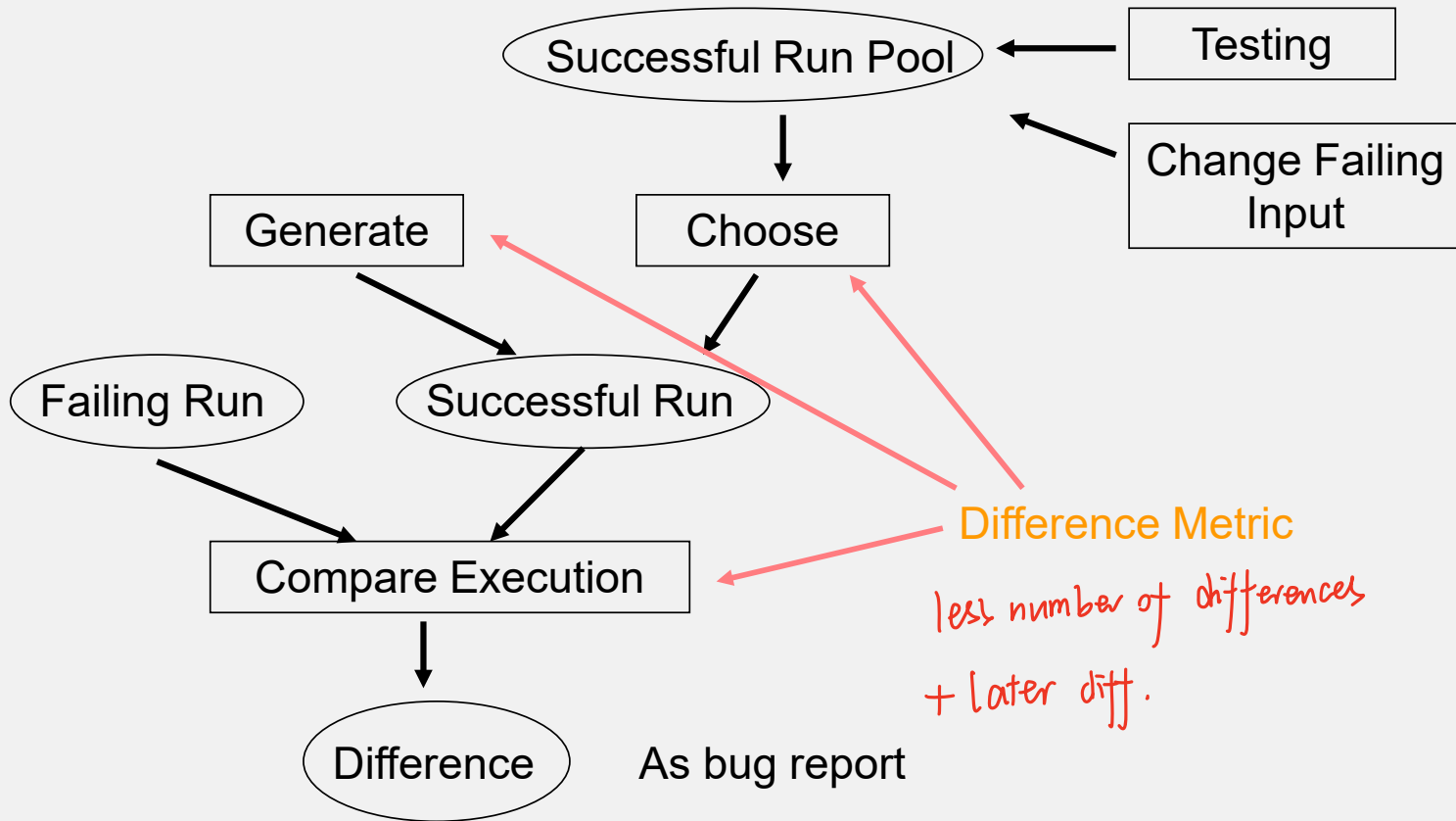
```
7.     if ( m == -1)
```

```
8.         ....
```

```
9. }
```

**Favor branches near to  
the observable error**

# FAULT LOCALIZATION



# SUSPICIOUS-NESS SCORE

- Given a test-suite  $T$

$$\text{Score}(s) = \frac{\frac{\text{fail}(s)}{\text{allfail}}}{\frac{\text{fail}(s)}{\text{allfail}} + \frac{\text{pass}(s)}{\text{allpass}}}$$

- $\text{fail}(s) \equiv \#$  of failing executions in which  $s$  occurs
- $\text{pass}(s) \equiv \#$  of passing executions in which  $s$  occurs
- $\text{allfail} \equiv \text{Total } \#$  of failing executions
- $\text{allpass} \equiv \text{Total } \#$  of passing executions
- $\text{allfail} + \text{allpass} = |T|$

# SUSPICIOUS-NESS SCORE

```
1  int is_upward( int inhibit, int up_sep, int down_sep){
2      int bias;
3      if (inhibit)
4          bias = down_sep; // bias= up sep + 100
5      else bias = up_sep ;
6      if (bias > down_sep)
7          return 1;
8      else return 0;
9  }
```

inhibit	up_sep	down_sep	Observed output	Expected Output	Result
1	0	100	0	0	pass
<b>1</b>	<b>11</b>	<b>110</b>	<b>0</b>	<b>1</b>	<b>fail</b>
0	100	50	1	1	pass
<b>1</b>	<b>-20</b>	<b>60</b>	<b>0</b>	<b>1</b>	<b>fail</b>
0	0	10	0	0	pass

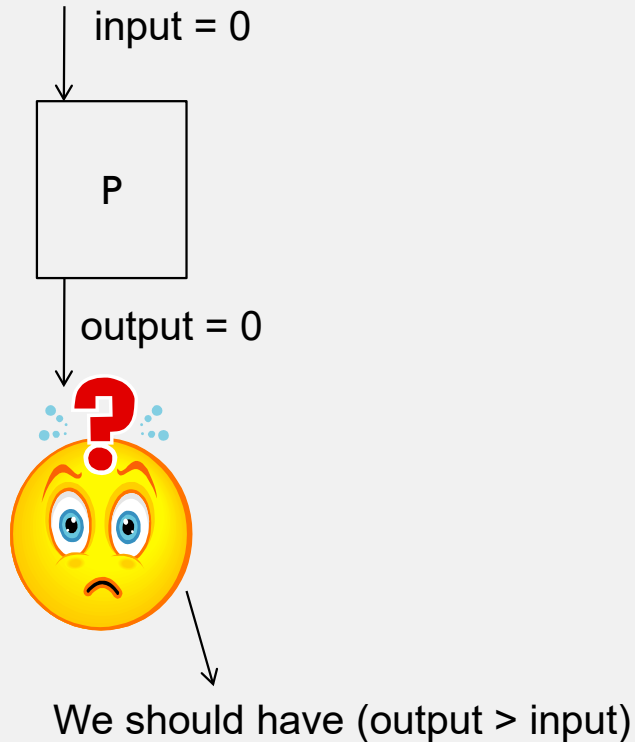
Line	Score	Rank
<b>4</b>	<b>0.75</b>	<b>1</b>
8	0.6	2
3	0.5	3
6	0.5	3
5	0	5
7	0	5



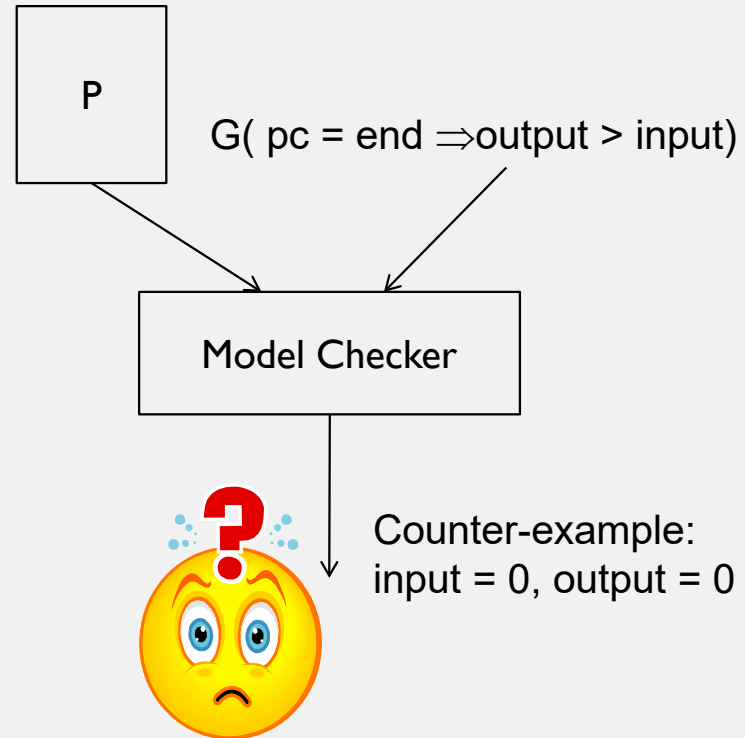
# BIG PICTURE – TESTING AND DEBUGGING

- Why test?
  - Feel good about the program you have written.
- How does it relate to fault localization?
  - Testing identifies which inputs we run the program against.
    - What is a good set of inputs to test?
  - Once you run the selected inputs, for some of them the output is unexpected.
    - These are the failing tests.
    - These are subjected to fault localization.

# DEBUGGING & VERIFICATION



**(a) Debugging**



**(b) Model Checking**

# VERIFICATION AND TESTING

- Model checking tries to check a specific property for all possible inputs
  - Same as checking a shallow property by exhaustive testing
  - Of course, the algorithms are more efficient than doing exhaustive testing.
- Testing checks an expected output for a specific program input.

**Materials on model checking are studied in CS4211.**