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
- Discussion of the thinking behind your course project
- Static analysis and vulnerability detection: Secure SE
- Test-suite estimation

- Today
  - **Software Debugging**

SOFTWARE CONSTRUCTION

- From a design model
  - In safety-critical domains – automotive, avionics.
  - D0 178C – software in airborne systems.
- Or, hand-constructed
  - Usual practice – audio, video and other domains.
  - UML models only for guidance.

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

ORGANIZATION

- Brief History of Debugging
- Dynamic checking of programs
  - Dynamic slicing
  - Relevant Slicing
  - 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

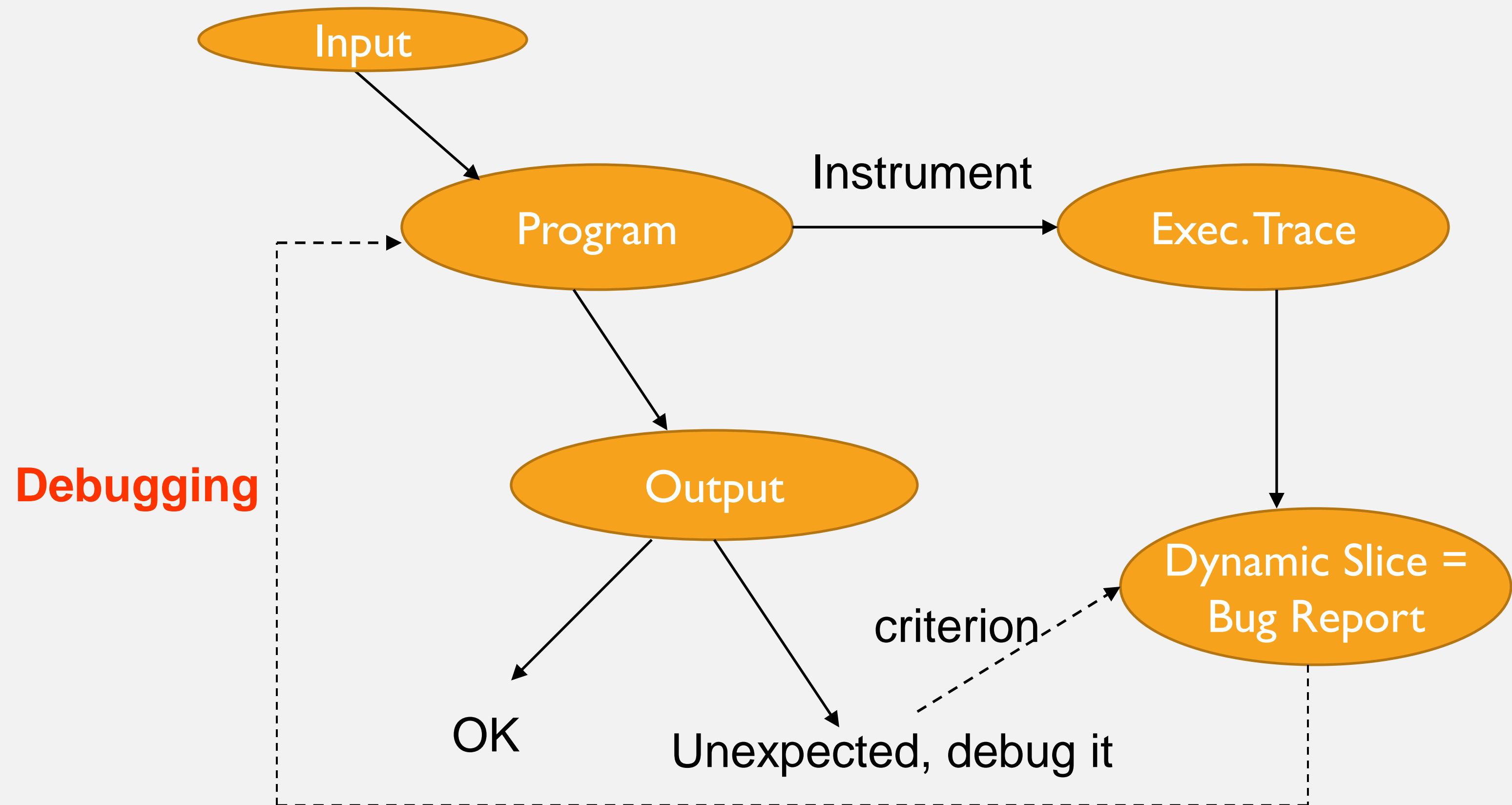
What about the current techniques, beyond breakpoints?

Let us review them first.

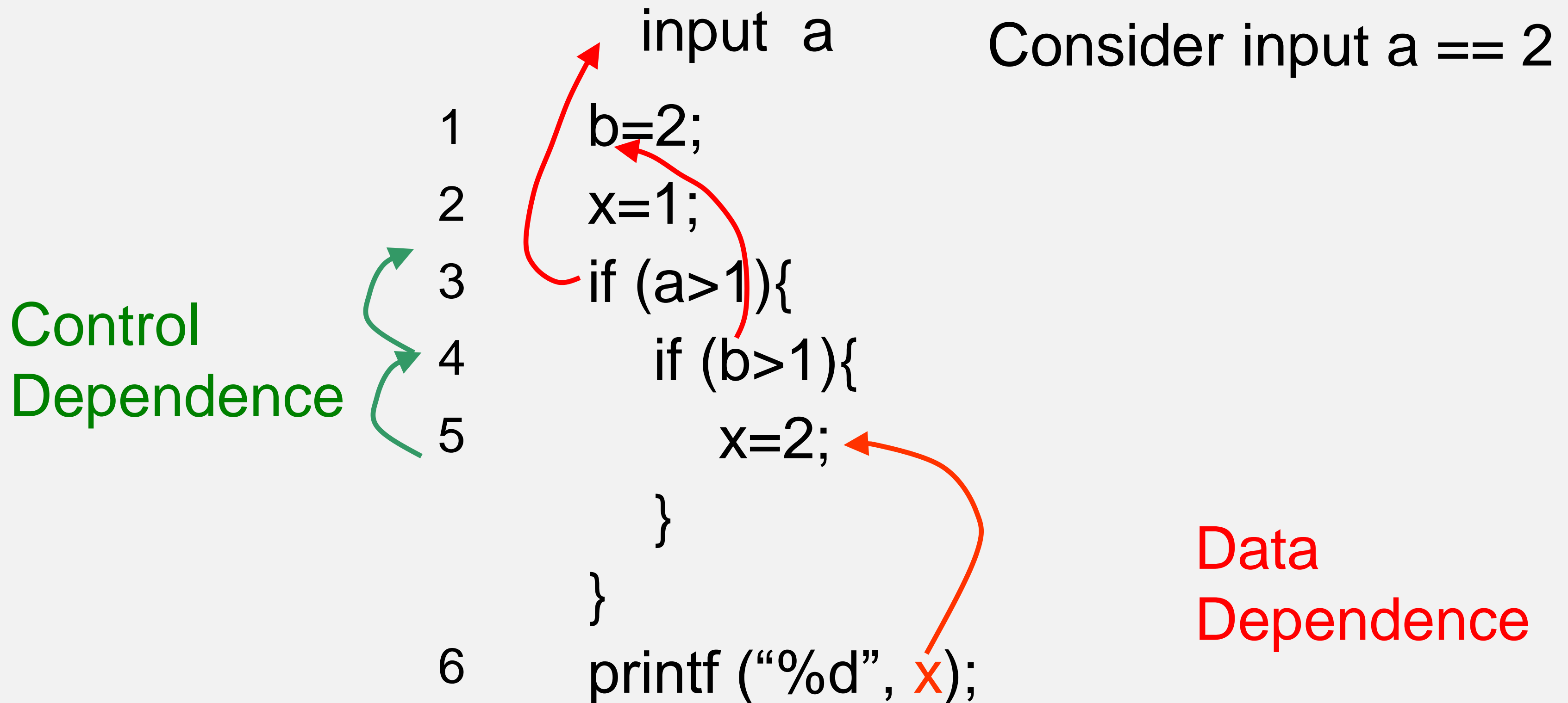
Any progress in 2002 – 2022?

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

DYNAMIC SLICING: A 1ST-GENERATION DEBUGGING AID



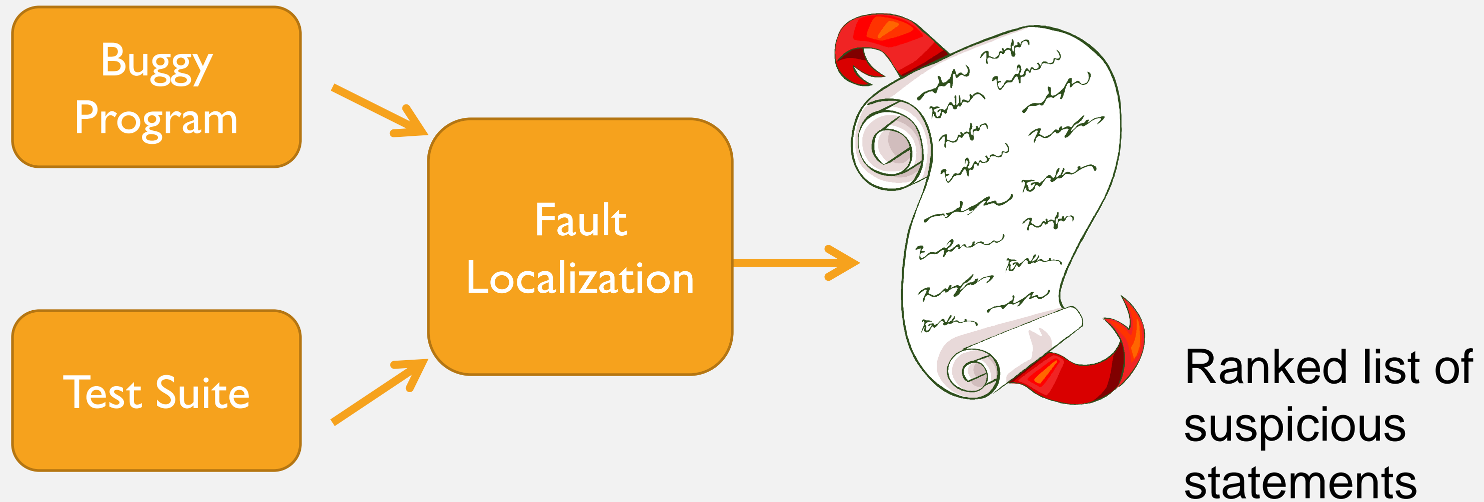
DYNAMIC SLICING



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!***

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

An example of scoring scheme [Tarantula]

RANKED BUG REPORT

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

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

  - fail(s) $\equiv$ # of failing executions in which s occurs
  - pass(s) $\equiv$ # of passing executions in which s occurs
  - allfail $\equiv$ Total # of failing executions
  - allpass $\equiv$ Total # of passing executions
    - allfail + 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	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	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	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$
Ample	$\left\lvert\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}\right\rvert$	Wong1	a_{ef}
Wong2	$a_{ef}-a_{ep}$		
Wong3	$a_{ef}-h$, where $h=\begin{cases}a_{ep} & \text{if } a_{ep}\le 2\\ 2+0.1(a_{ep}-2) & \text{if } 2<a_{ep}\le 10\\ 2.8+0.001(a_{ep}-10) & \text{if } a_{ep}>10\end{cases}$		
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	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
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

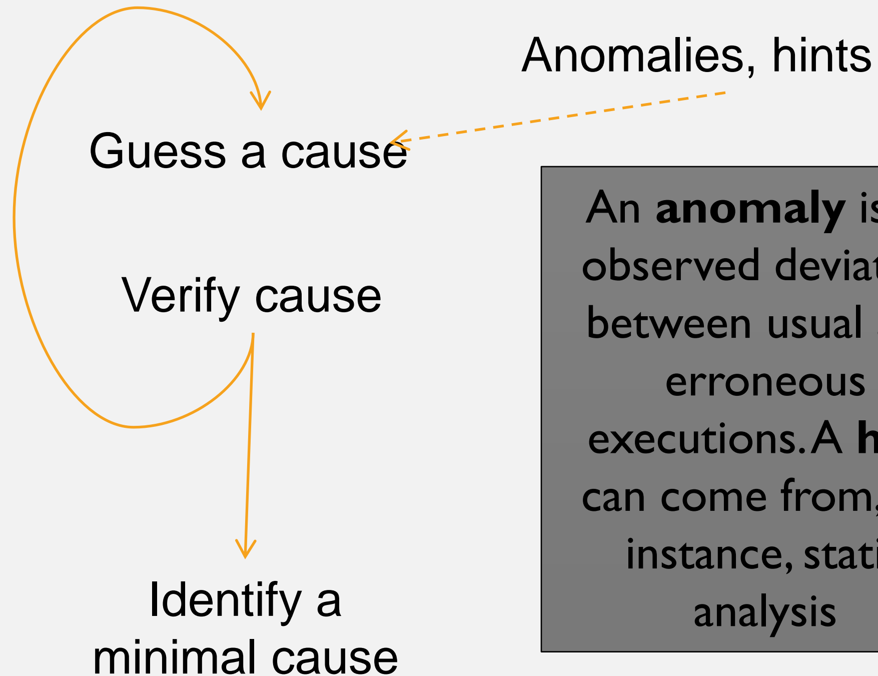
$$\text{Score}(s)=\frac{\text{fail}(s)}{\sqrt{\text{allfail}*(\text{fail}(s)+\text{pass}(s))}}$$

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

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

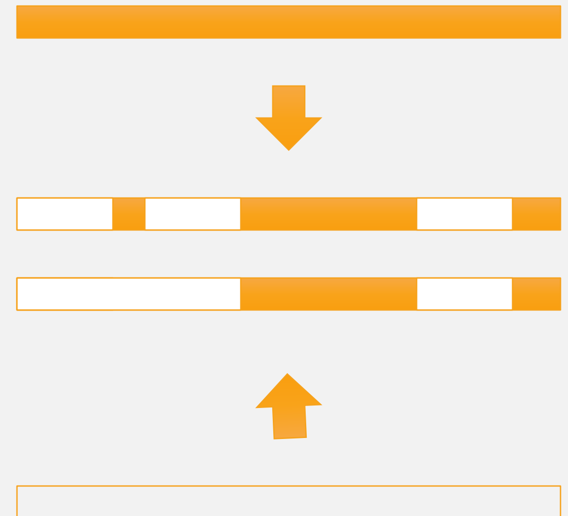


An **anomaly** is an observed deviation between usual and erroneous executions. A **hint** can come from, for instance, static analysis

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

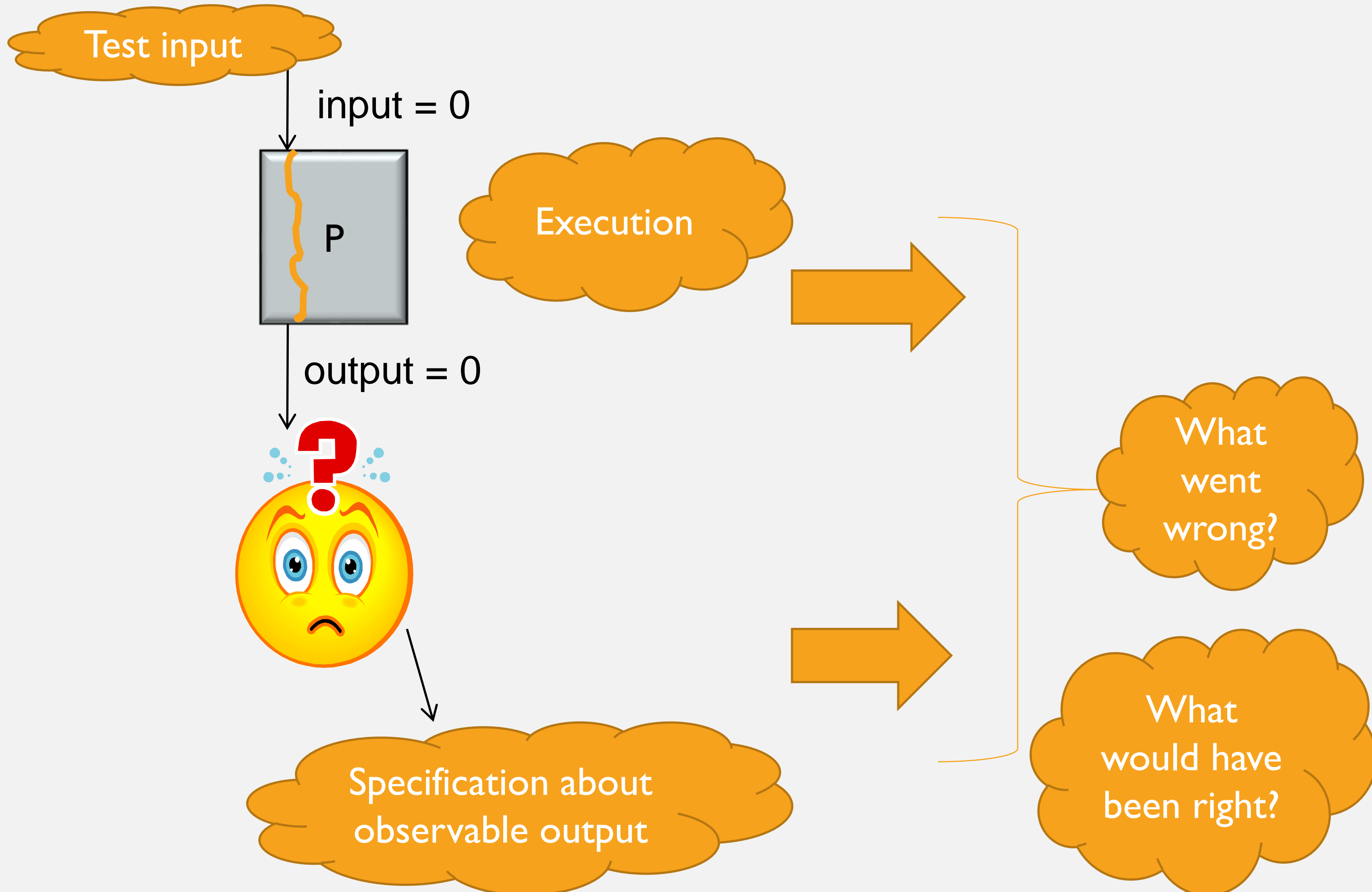
ISOLATING FAILURE CAUSES A LA DELTA DEBUGGING

- 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

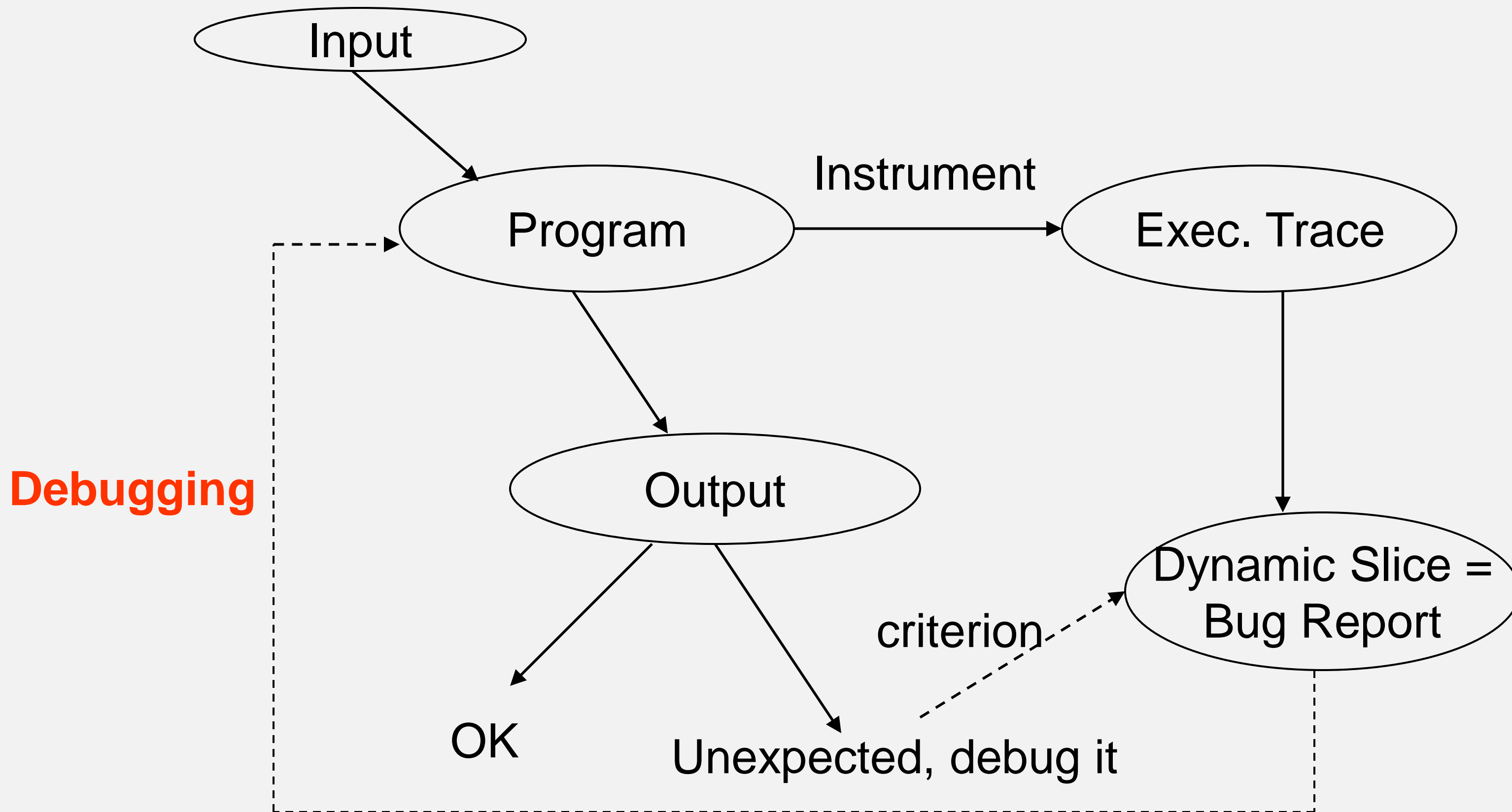
THE ROLE OF SPECIFICATIONS



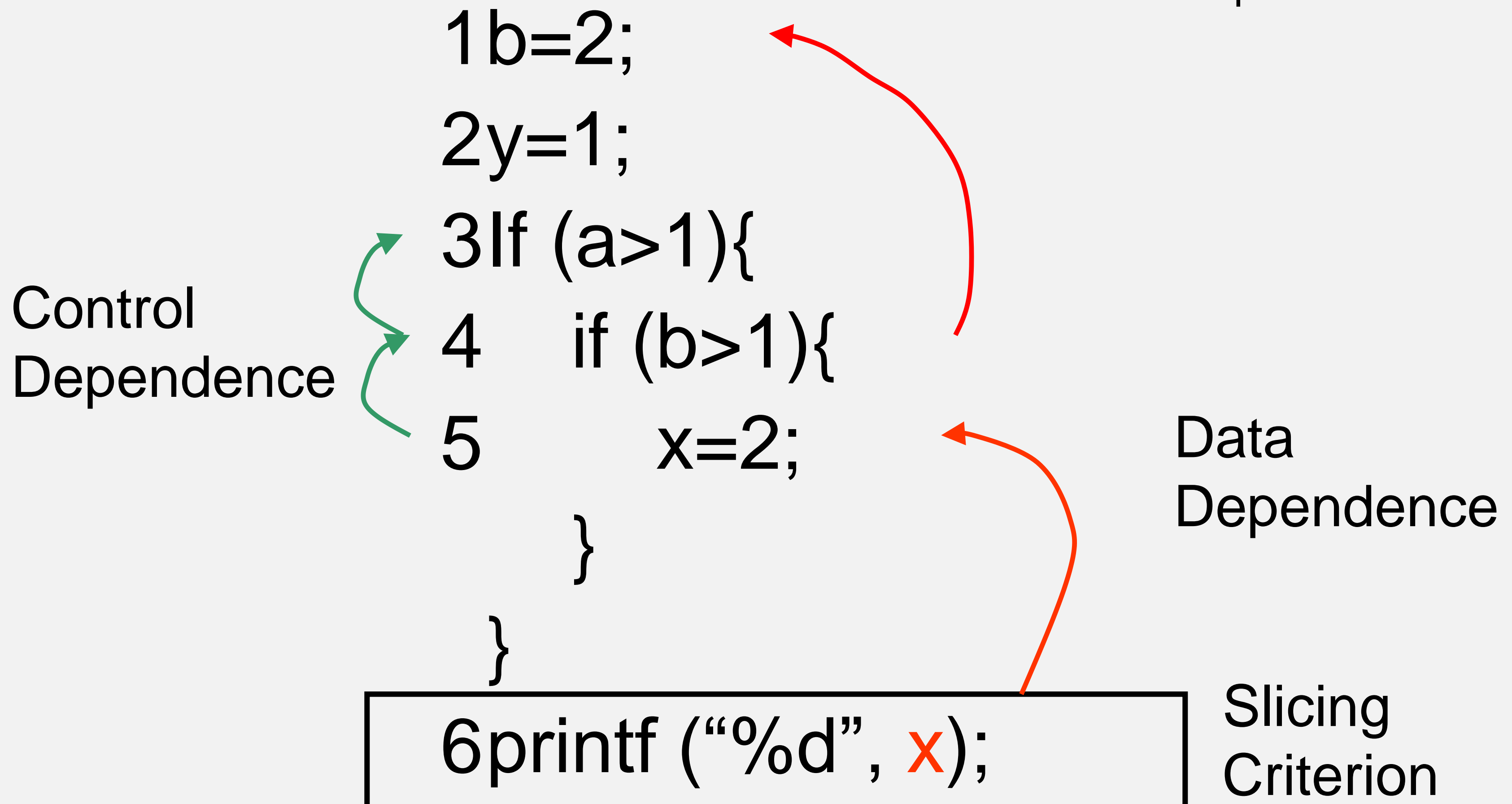
ORGANIZATION

- Brief History of Debugging
- Dynamic checking of programs
  - Dynamic slicing
  - Relevant slicing
  - Fault Localization

DYNAMIC SLICING FOR DEBUGGING



DYNAMIC SLICING



DYNAMIC SLICE

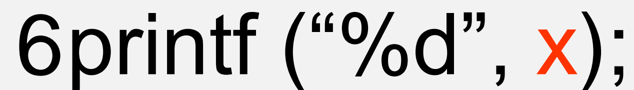
- Set slicing criterion
  - (Variable v at first instance of line 70)
  - The value of variable v at first instance of line 70 is unexpected.
- Dynamic slice
  - Closure of
    - Data dependencies &
    - Control dependencies
  - from the slicing criterion.

STATIC VS DYNAMIC SLICING

- Static Slicing
  - source code
  - statement
  - static dependence
- Dynamic Slicing
  - a particular execution
  - statement instance
  - dynamic dependence

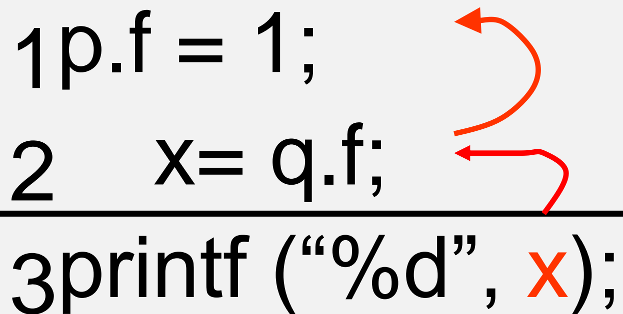
STATIC VS DYNAMIC SLICING

```
1 b=1;
2 If (a>1)
3     x=1;
4 else
5     x=2;
```



Slicing Criterion

STATIC VS DYNAMIC SLICING



Slicing Criterion

p and q point to the same object?

- Static points-to analysis is always conservative

RELEVANT SLICING

input: a=2

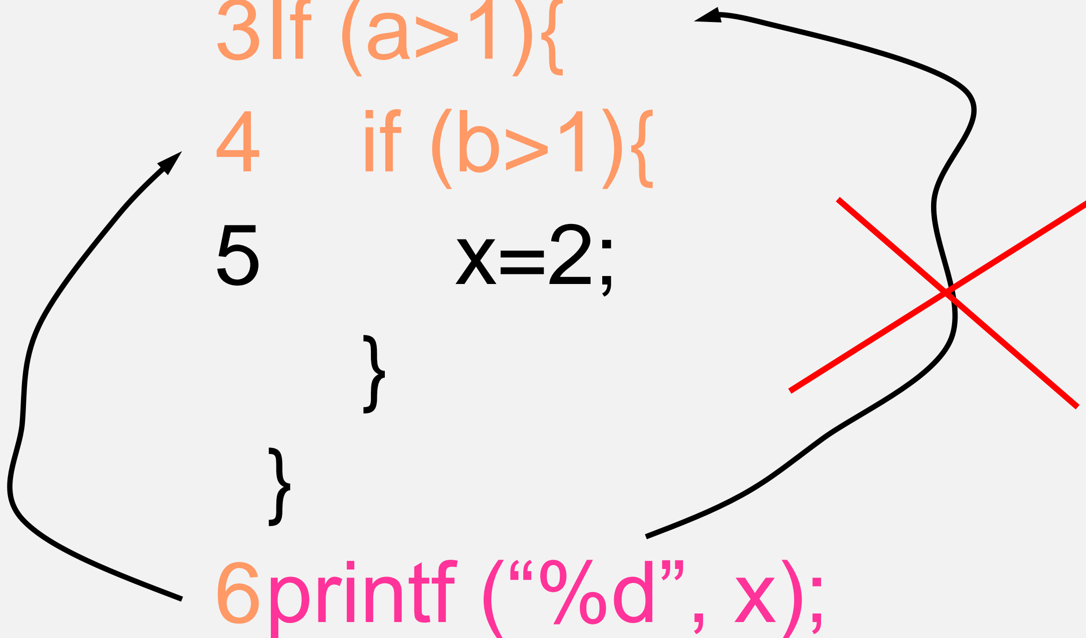
Source of Failure

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);
```

POTENTIAL DEPENDENCE



RELEVANT SLICE

input: a=2

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

Potential Dependence

Dynamic Data Dependence

PROGRAM SLICE

Static	Dynamic	Relevant
1		1
2	2	2
3		
4		4
5		
6	6	6

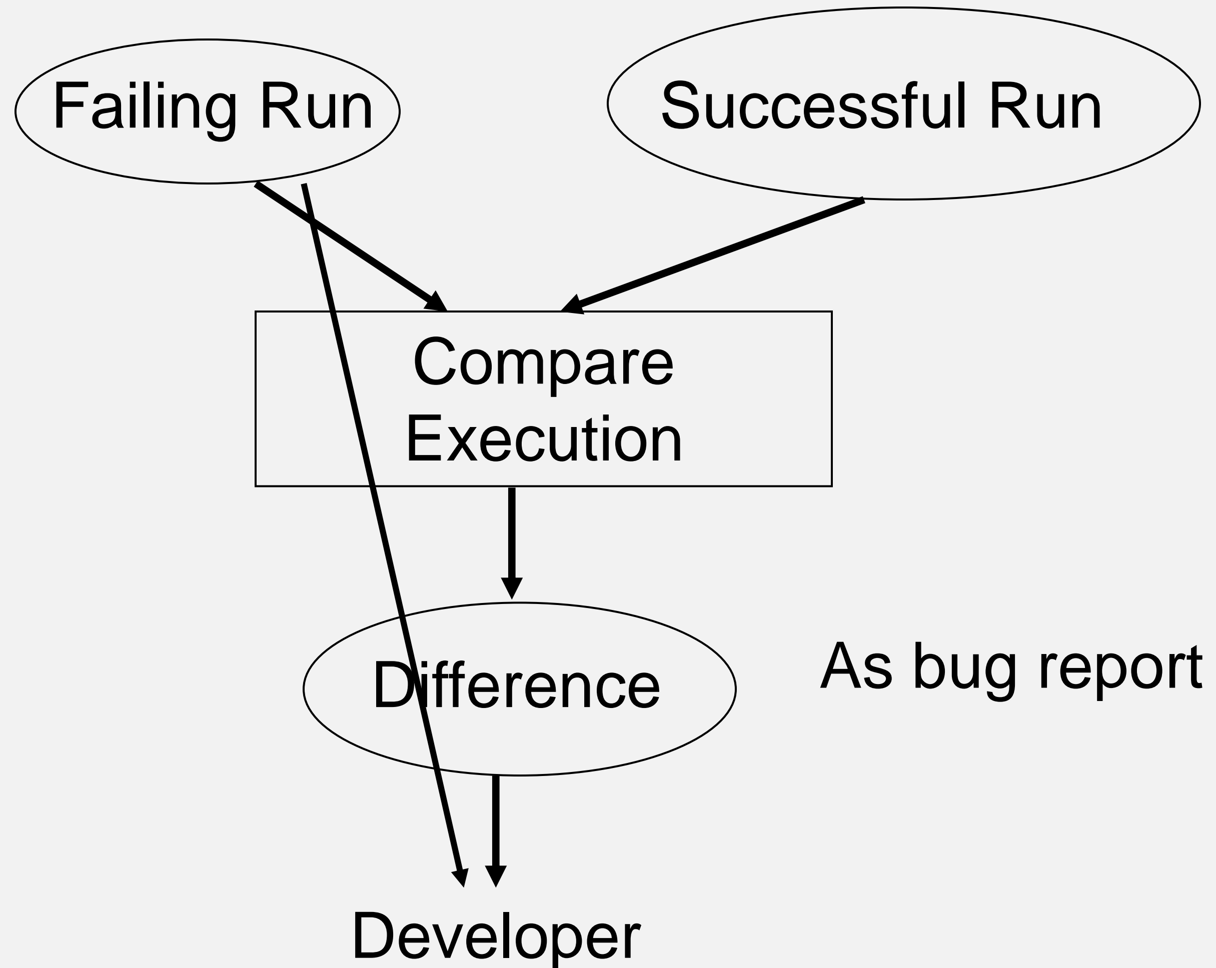
```
1 b=1;
2 x=1;
3 If (a>1){
4     if (b>1){
5         x=2;
      }
  }
6 printf ("%d", x);
```

input: a=2

ORGANIZATION

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

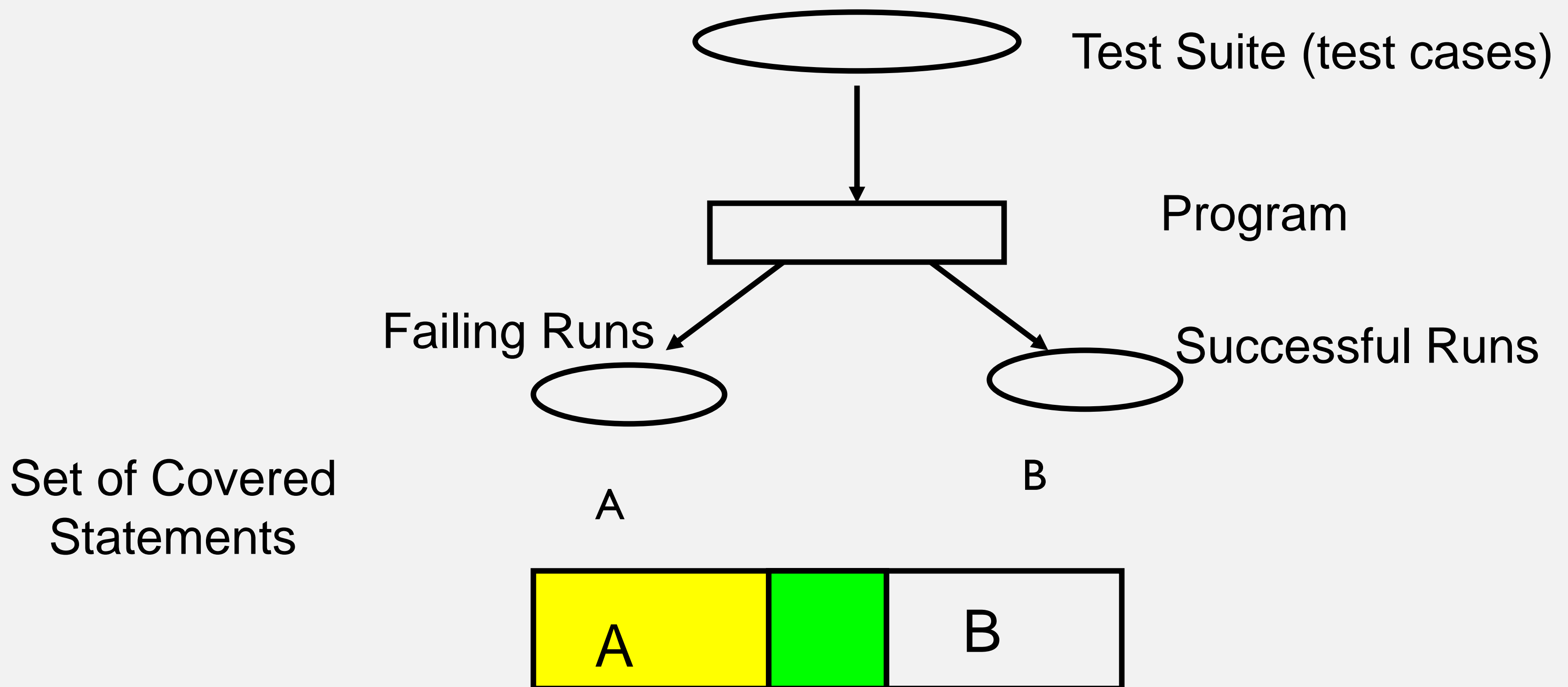
FAULT LOCALIZATION: OVERVIEW



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

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

if (x>=0)

FAULT LOCALIZATION - BRANCHES

```
1.   v=0;
2.   if (x>0)


4.   else
5.      u=v;
6. printf("%d",u);
```

Failing run, x=0

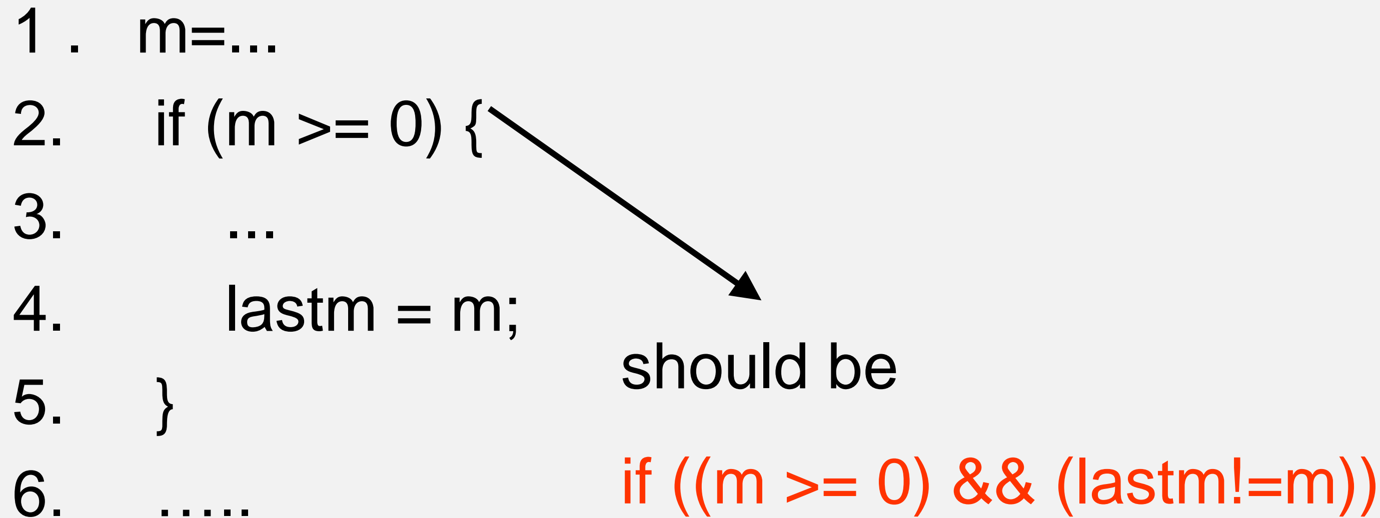
```
1.   v=0;
2.   if (x>0)
3.      u=5;



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

Successful run, x=1

COMPARING EXECUTIONS



COMPARING EXECUTIONS

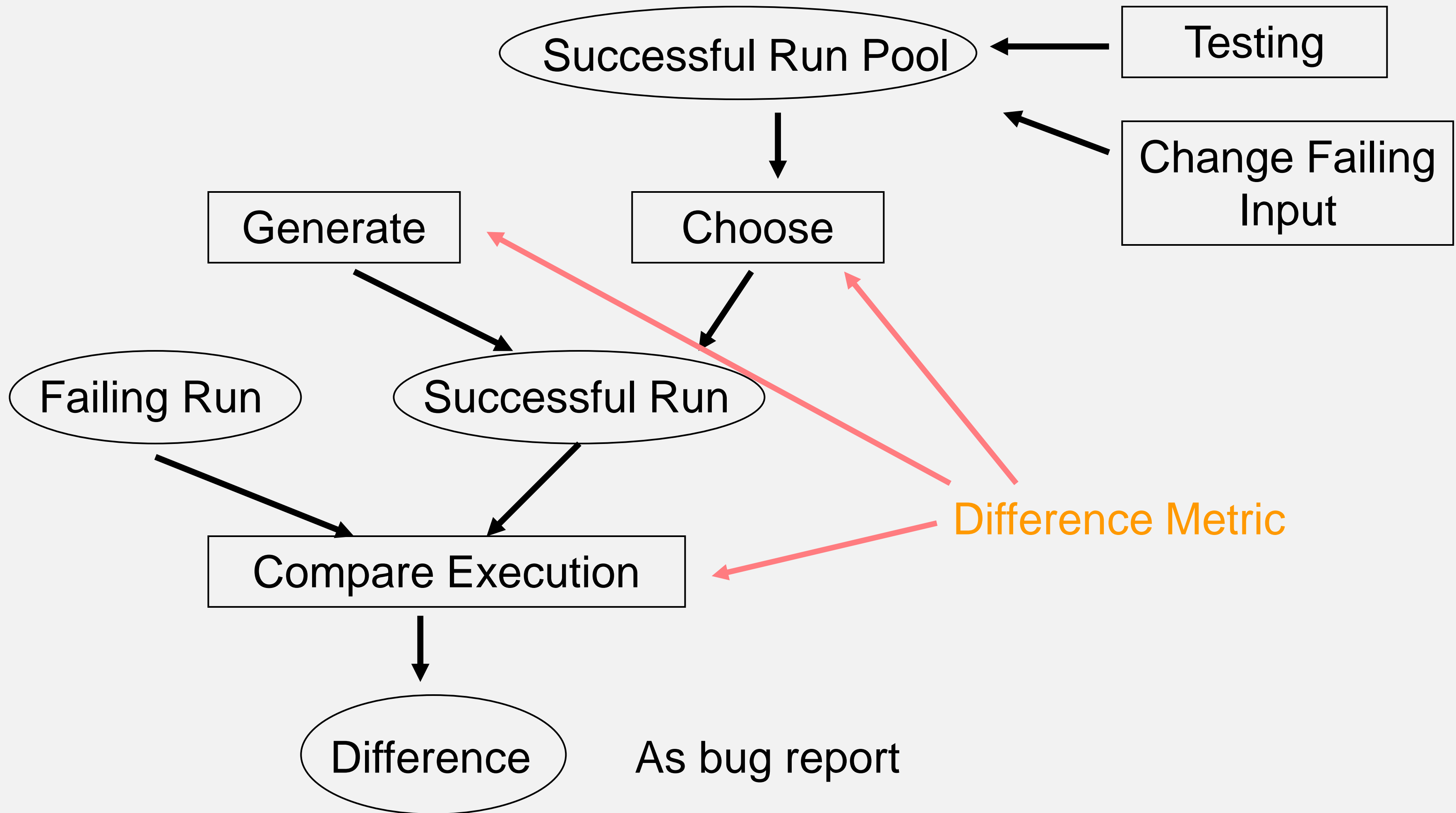
```
1 .   m=...
2.      if (m >= 0) {
3.          ...
4.          lastm = m;
5.      }
6.      .....
```

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 π

```
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 π1

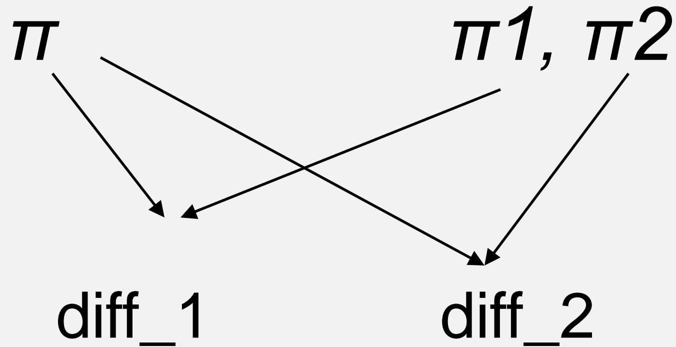
SET OF STATEMENTS

- S = Set of statements executed in π
  - {1,3,5,6,7,10}
- S1 = Set of statements executed in $\pi 1$
  - {1,3,4,5,6,9,10}
- If π is faulty and $\pi 1$ is passing
  - Bug report = S – S1 = {4,7}
- Choice of the execution run to compare with is very important.

ANOTHER DIFFERENCE METRIC

Failing Run

Successful Runs



Compare

- Number of Branches
- Location of Branches

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);
```

COMPARE *CORRESPONDING* STATEMENT INSTANCES

```
1.  while (a){
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)
```

1st Loop Iteration

2nd Loop Iteration

3rd Loop Iteration

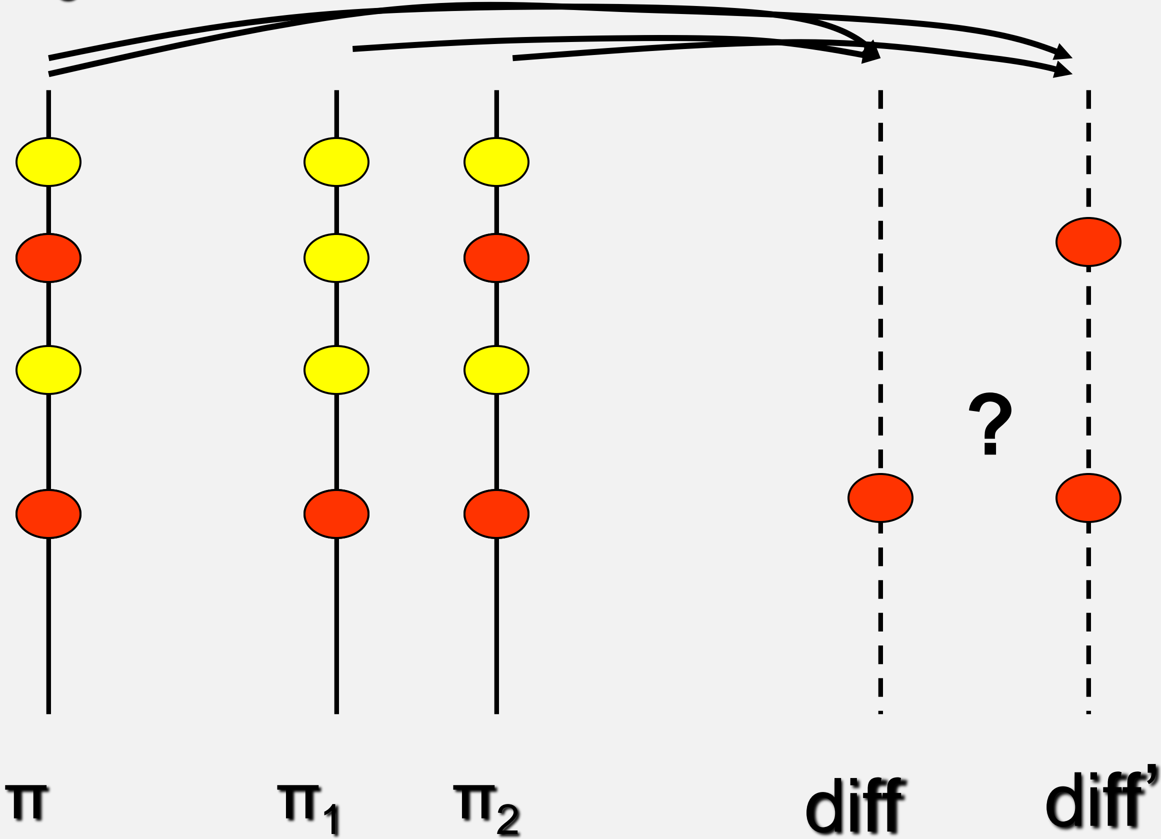
Use control dependencies!

FORMAL NOTION OF ALIGNMENT

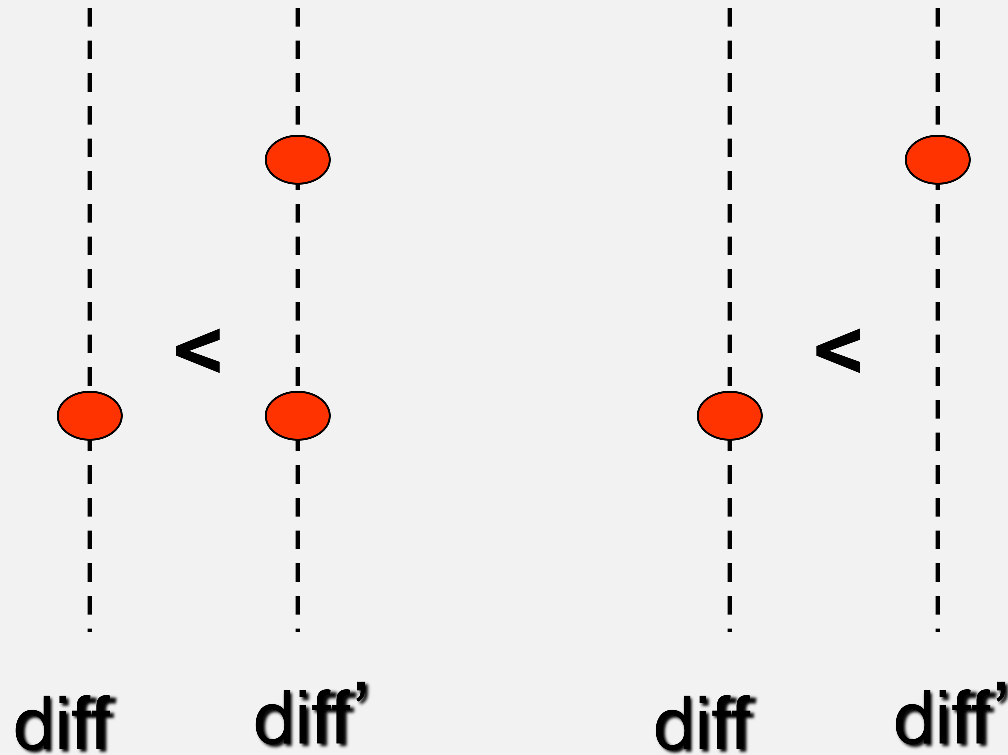
For any pair of event e in run x and event e0 in run y, we define align(e, e0) = true (e and e0 are aligned) iff.

- stmt(e) = stmt(e0), and
- either
  - e, e0 are the first events appearing in runs x, y or
  - align(dep(e, x), dep(e0, y)) = true.
    - 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

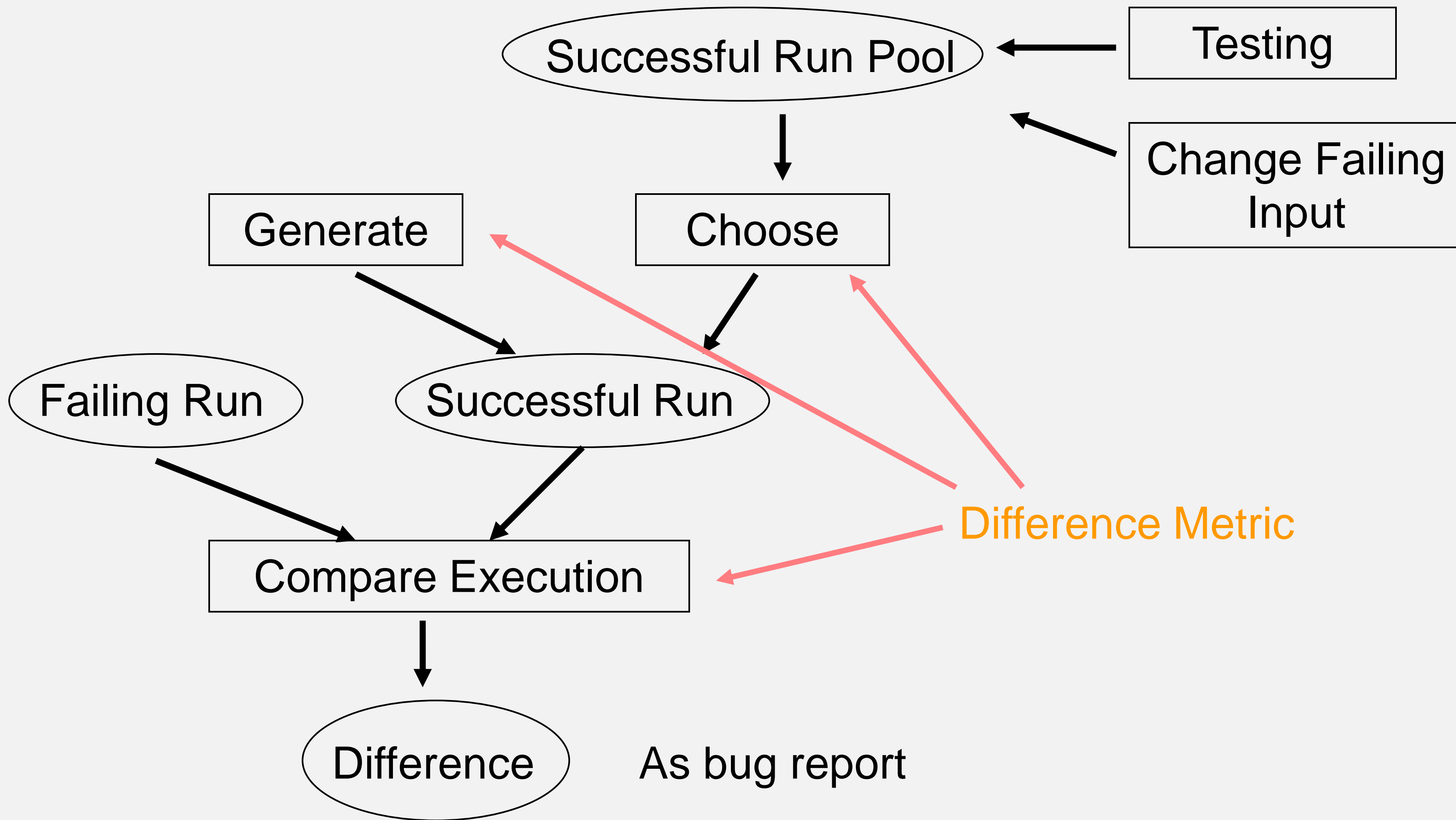
```
1.  int main(int argc, char **argv)
2.       if (argc < 3 ){
3.           printf("parameter error\n");
4.           return 0;
5.       }

6.        ....
7.       if ( m == -1)
8.           ....
9.   }
```

check the input

Favor branches near to the observable error

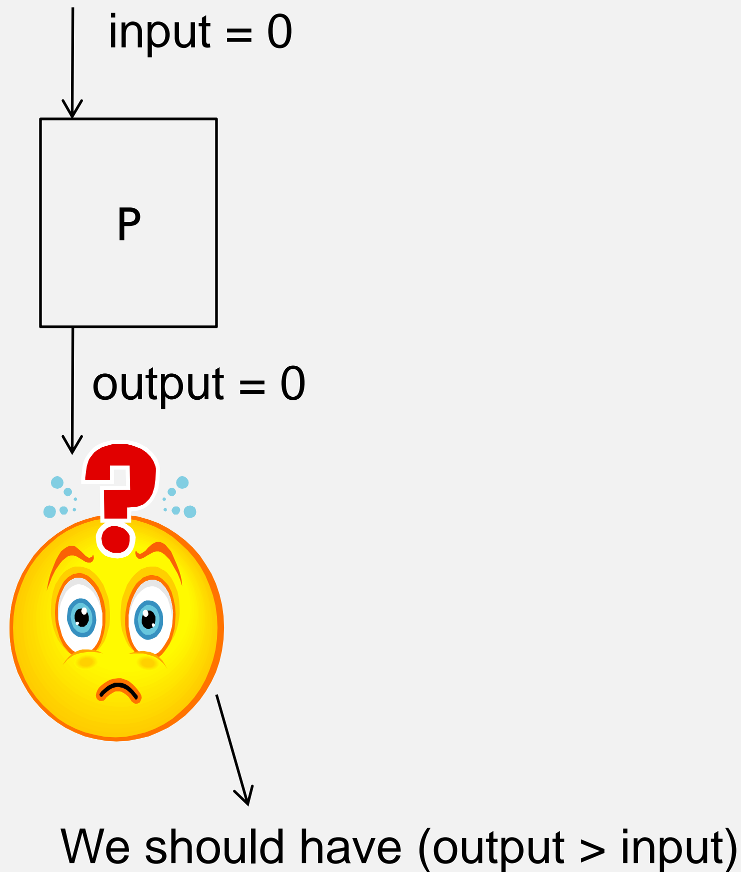
FAULT LOCALIZATION – IN SUMMARY

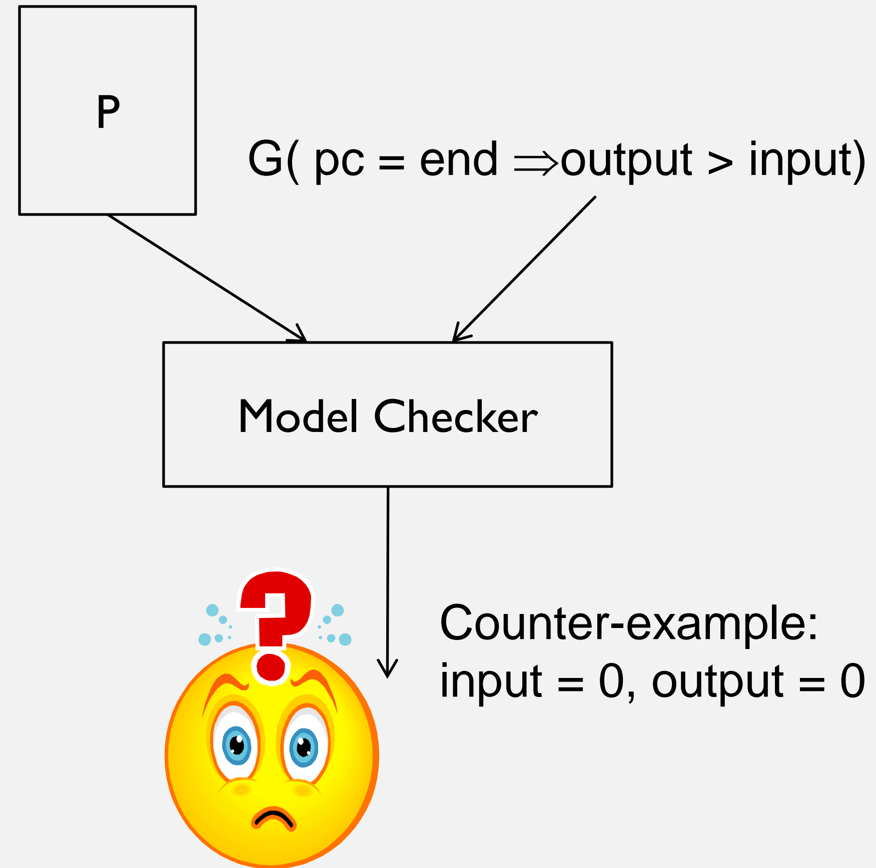


BIG PICTURE – TESTING AND DEBUGGING

- 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.