Software Validation
CS 4271 Lecture 11, 12

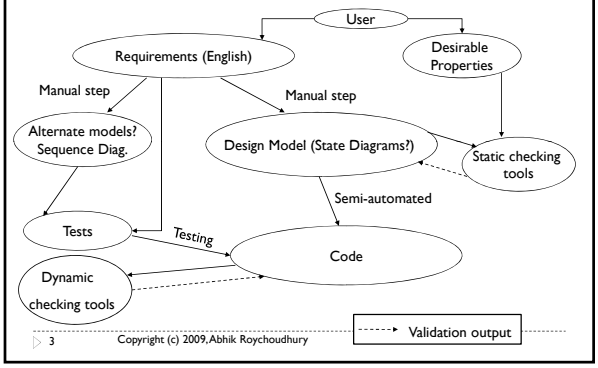
Abhik Roychoudhury
National University of Singapore

http://www.comp.nus.edu.sg/~abhik/

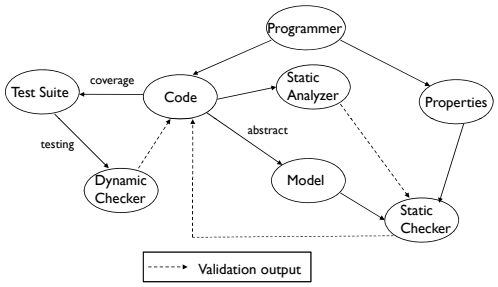
Software construction

- 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.

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?

You were right: There's a needle in this haystack...

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

- Dynamic checking of programs
  - Dynamic slicing
  - Hierarchical slicing
  - Fault Localization
  - Directed testing
- Static checking of programs – Not covered.
  - Predicate abstraction
  - Abstraction refinement

What is dynamic checking?

- Check program executions, not source code.
- How to generate program executions?
  - Testing (coverage based)
  - Testing (specification based)
- How to check program executions
  - Data and control dependencies (slicing)
  - By comparing against other program executions (fault localization).

SW Debugging: Social aspects

Software-controlled devices are ubiquitous ---
automotive control, avionics control and consumer electronics
Many of these software are safety-critical
$\Rightarrow$ 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**
  - **$40K salary per year**

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

$$\frac{65{,}000}{44} \textbf{ weeks = 1477 weeks = } \frac{1477}{50} \approx \textbf{ 30 years = \$1.2 M}$$

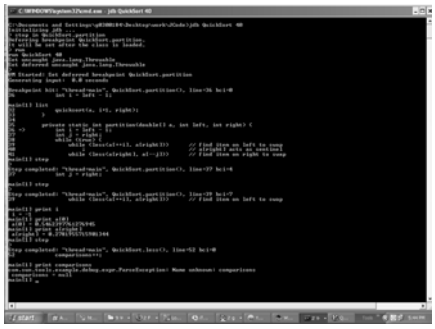
SW Debugging: tools

"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)

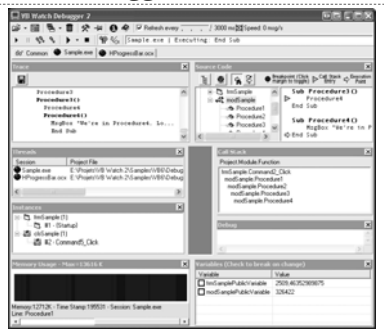
-> Need methods and tools to trace back to the root cause of bug from the manifested error

-> What about the current tools?

jdb on windows XP



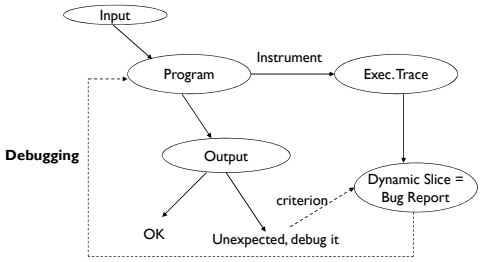
VB watch debugger



So, what did we see?

- Command line tool for Java
  - 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.

Dynamic Slicing for Debugging



Dynamic Slicing

Consider input a == 2

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

Control Dependence

Data Dependence

Slicing Criterion

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.

A[i] := 1;
…
U := A[j]

→ Do we consider this data dependence edge ?

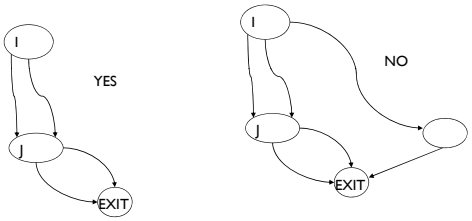
→ Remember that the slicing is for an input, so the addresses are resolved

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

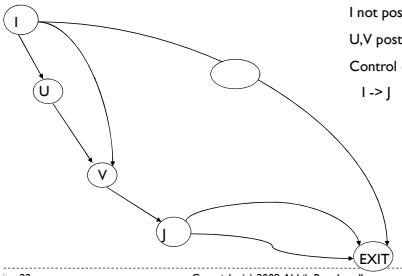
Static Control dependencies

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 -> J

Dynamic control dependencies

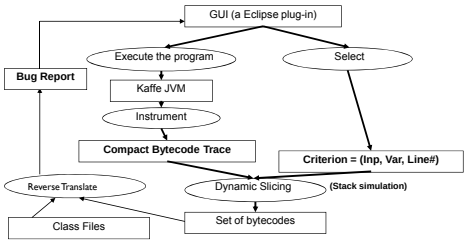
- X is dynamically control dependent on Y if
  - Y occurs before X in the execution trace
  - X's stmt. is statically control dependent on Y's stmt.
  - No statement Z between Y and X is such that X's stmt. is statically control dependent on Z's stmt.
- Captures the intuition:
  - What is the nearest 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

Jslice: a dynamic slicing tool



http://jslice.sourceforge.net

Issues for such a slicing tool

- Online trace compression – beyond conventional string compression.
  - Full trace is never stored.
- Program dependence analysis on compressed trace – no decompression.
- Analysis at low-level (byte-code) to support third-party software.
  - Managing stack architecture.

Organization

- Dynamic checking of programs
  - Dynamic slicing
  - Hierarchical slicing
  - Fault Localization
  - Directed testing
- Static checking of programs
  - Predicate abstraction
  - Abstraction refinement

Problem with dynamic slicing

- Huge overheads
  - Backwards slicing requires trace storage.
  - Jslice tool for Java
    - Online trace compression & traversal
    - http://jslice.sourceforge.net
- Dynamic Slice is still too large …
  - … for human comprehension
  - Now

An example

```
1   public static void main(String[] args) {
      …….
2.    init( db );
3     operate( db );
4.    output ( db )
5.    return;
}
```

SPECJVM DB program

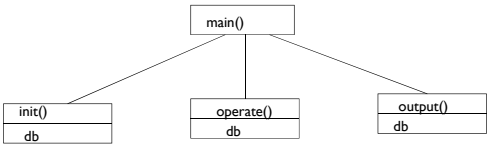
```
init( .. db) {
    db= ..
    ….
}
```

```
operate (… db ) {
    db =..
      …
}
```

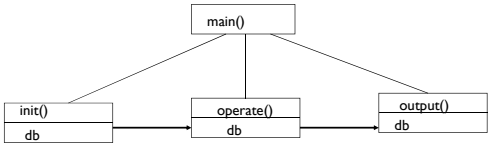
```
output (db) {
    ……
    print(db...);
}
```


Divide trace into phases

```
1   public static void main(String[] args) {
      …….
2.    init( db );
3     operate( db );
4.    output ( db );
5.    return;  }
```



Report inter-phase dependencies



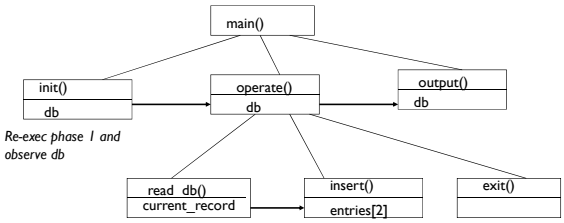
Intra-phase control and data dependencies are suppressed.

Inter-phase dep. form input-output relationships.

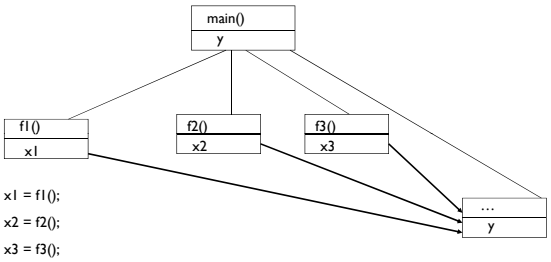
Programmer zooms into …

… one phase by inspecting the phase outputs

-> (may/may not involve re-executing program)



Parallel Dependence Chains



x1 = f1();

x2 = f2();

x3 = f3();

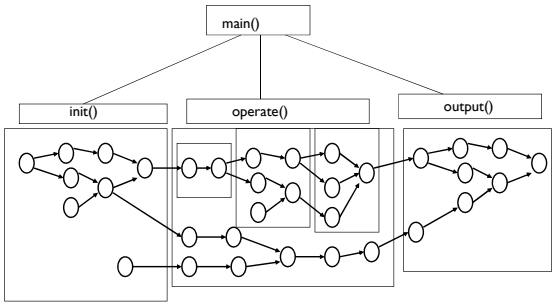
y = x1 + x2 + x3;

print y — Criterion

Hierarchical dynamic slicing

- Compute "phases" of an exec. trace
  - Control structure boundaries
- Augment dynamic slicing algorithm
  - Mark inter-phase dependencies
  - Compute only reachable nodes from selected inter-phase dependency.
- Programmer intervention
  - Select the first suspicious inter-phase dep.
  - Comprehension guides computation.

In action …



Beyond Dynamic Slices

- If dynamic slice computation and traversal becomes manageable
  - We can look beyond dynamic slices.
  - We can look at errors which are not captured in dynamic slices.

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;
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

Relevant Slicing

```
1 b=10;
2 x=1;
3 If (a>1){
4    if (b>1){
5       x=2;
     }
  }
6 printf ("%d", x);
```

Relevant Slicing

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

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

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

Relevant Slice

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

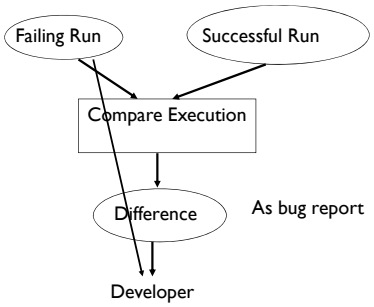
Organization

- Dynamic checking of programs
  - Dynamic slicing
  - Hierarchical slicing
  - Fault Localization
  - Directed testing
- Static checking of programs
  - Predicate abstraction
  - Abstraction refinement

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 an OK trace to localize the source of error.

Fault Localization: overview



Comparing executions

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

should be

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

Comparing executions

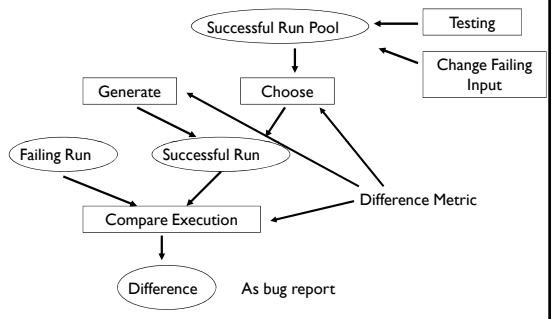
```
1 .  m=...
2.   if (m >= 0) {
3.       ...
4.       lastm = m;
5.   }
6.   …..
```

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 π

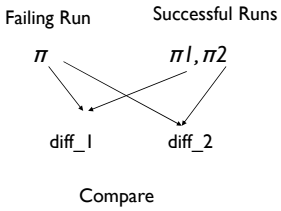
```
1.  if (a)
2.      i = i + 1;
3.  if (b)
4.      j = j + 1;
5.  if (c)
6.      if (d)
7.          k = k + 1;
8.      else
9.          k = k + 2;
10. printf("%d", k);
```

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 π1
  - {1,3,4,5,6,9,10}
- If π is faulty and π1 is OK
  - Bug report = S – S1 = {4,7}
- Choice of the execution run to compare with is very important.

Another difference metric



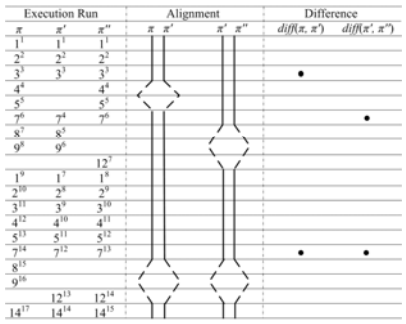
- Number of Branches
- Location of Branches

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);
```


Trace alignment and differences



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

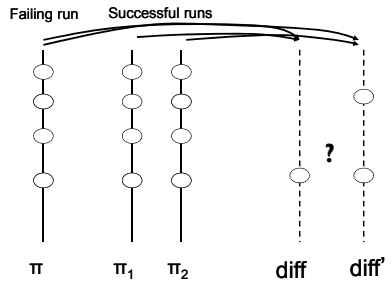
Use control dependencies!

Formal notion of Alignment

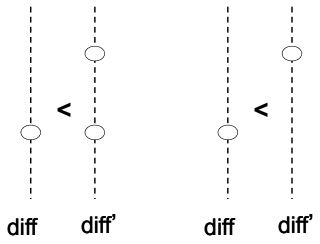
For any pair of event e in run x and event e0 in run y, we define align(e, e0) = true (e and e0 are aligned) iff.

- 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.

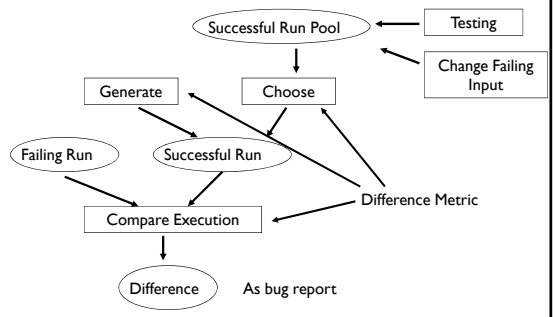
Comparison of differences



Comparison of differences



Fault localization – In summary



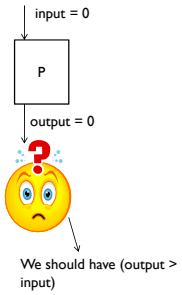
Organization

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

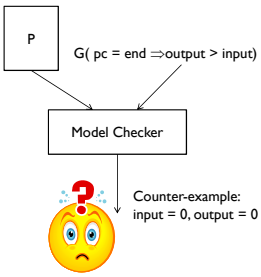
Big picture – Testing and Debugging

- 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.

Big picture – Debugging & MC



(a) Debugging

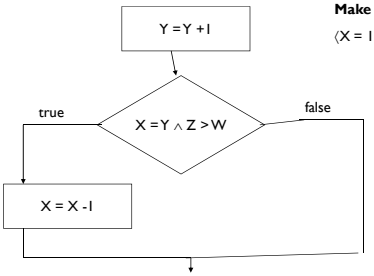


(b) Model Checking

Common terminology

- Test case
  - A test input (or its execution trace)
- Test suite
  - Set of test cases
- Test purpose
  - A formal specification to guide testing
    - e.g. a regular expression which the test case should satisfy
- Coverage criterion
  - A guide to exhaustively cover program structure.
    - e.g. Statement coverage, Cond. coverage, Path coverage.

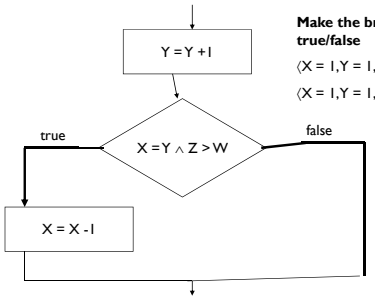
Statement coverage



Make the branch condition true

⟨X = 1, Y = 1, Z = 2, W = 1⟩

Edge coverage



Make the branch condition true/false

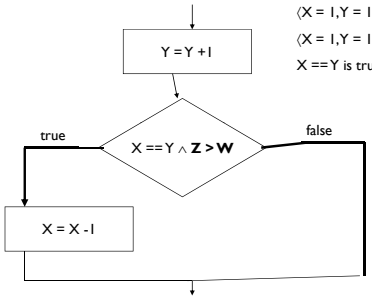
⟨X = 1, Y = 1, Z = 2, W = 1⟩

⟨X = 1, Y = 1, Z = 2, W = 2⟩

Condition coverage

- For each executable condition c
  - Check whether it can be both true or false
    - c could be unsatisfiable or valid in all pgm. executions
  - For all such conditions c, c should be true in at least one test in the test suite, and c should be false in at least one test in the test suite.

Condition coverage

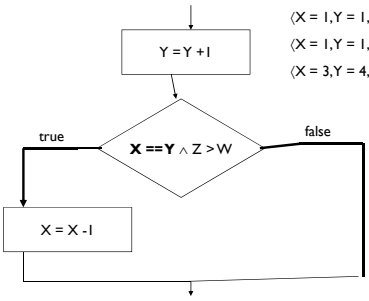


⟨X = 1, Y = 1, Z = 2, W = 1⟩

⟨X = 1, Y = 1, Z = 2, W = 2⟩

X == Y is true in both the test cases

Condition coverage



⟨X = 1, Y = 1, Z = 2, W = 1⟩

⟨X = 1, Y = 1, Z = 2, W = 2⟩

⟨X = 3, Y = 4, Z = 7, W = 5⟩

Path coverage

- Cover all paths in the program
  - Unboundedly many, unless loops can be bounded.
  - Lot of infeasible paths i.e. paths which do not form execution trace for any input.
    - Infeasible path detection will help test-suite construction.
- A technique to help exercise new paths with new tests
  - Attempts to achieve path coverage
  - Basic idea: concrete and *symbolic* execution at the same time.

Directed testing

- Start with a random input I.
- Execute program P with I
  - Suppose I executes path p in program P.
  - While executing p, collect a symbolic formula f which captures the set of all inputs which execute path p in program P.
  - **f is the path condition of path p traced by input i.**
- Minimally change f, to produce a formula f1
  - Solve f1 to get a new input I1 which executes a path p1 different from path p.

Example program

- if (Climb)
  - separation = Up;
- else
  - separation = Up + 100;
- if (separation > 150)
  - upward = 1;
- else
  - upward = 0;
- if (upward >0)
  - printf("Upward");
- else
  - printf("Downward);

Start with random input

(Climb == 0, Up == 457)

Example program

- if (Climb)
  - separation = Up;
- else
  - separation = Up + 100;
- if (separation > 150)
  - upward = 1;
- else
  - upward = 0;
- if (upward >0)
  - printf("Upward");
- else
  - printf("Downward);

Climb == 0 $\wedge$

(Up + 100 > 150) $\wedge$

upward > 0

Generating new tests

- The path condition calculated
  - Climb ==0 $\wedge$ Up + 100 > 150 $\wedge$ upward > 0
- Minimally modify the condition
  - Climb ==0 $\wedge$ Up + 100 > 150 $\wedge$ $\neg$(upward > 0)
- Corresponding to the path …

Infeasible path!!

- if (Climb)
  - separation = Up;
- else
  - separation = Up + 100;
- if (separation > 150)
  - upward = 1;
- else
  - upward = 0;
- if (upward >0)
  - printf("Upward");
- else
  - printf("Downward);

Climb == 0 $\wedge$

(Up + 100 > 150) $\wedge$

$\neg$ upward > 0

Generating new tests

- The path condition calculated
  - Climb ==0 $\wedge$ Up + 100 > 150 $\wedge$ upward > 0
- Minimally modify the condition
  - Climb ==0 $\wedge$ Up + 100 > 150 $\wedge$ $\neg$(upward > 0)
  - Corresponding to infeasible path!
- Modify a bit more
  - Climb == 0 $\wedge$ $\neg$ (Up + 100 > 150)
  - Corresponding to the path …

Feasible path

- if (Climb)
  - separation = Up;
- else
  - separation = Up + 100;
- if (separation > 150)
  - upward = 1;
- else
  - upward = 0;
- if (upward >0)
  - printf("Upward");
- else
  - printf("Downward);

Climb == 0 $\wedge$

$\neg$ (Up + 100 > 150)

Generating new tests

- The path condition calculated
  - Climb ==0 $\wedge$ Up + 100 > 150 $\wedge$ upward > 0
- Minimally modify the condition
  - Climb ==0 $\wedge$ Up + 100 > 150 $\wedge$ $\neg$(upward > 0)
  - Corresponding to infeasible path!
- Modify a bit more
  - Climb == 0 $\wedge$ $\neg$ (Up + 100 > 150)
  - Solve to get another test input
    - Climb == 0, Up == 0
- Continue in this fashion.

Path condition computation

```
1 input x, y, z;
2 if (y > 0){
3    z = y * 2;
4     x = y - 2;
5     x = x - 2; }
6 if (z == x){
7     output("How did I get here");
}
```

Path condition computation

```
1 input x, y, z;
2 if (y > 0){
3    z = y * 2;
4     x = y - 2;
5     x = x - 2; }
6 if (z == x){
7     output("How did I get here");
}
```

Line#	Assignment store	Path cond.
1	{}	true
2	{}	y > 0
3	{(z, 2*y)}	y > 0
4	{(z,2*y), (x, y-2)}	y > 0
5	{(z,2*y), (x, y-4)}	y > 0
6	{(z,2*y), (x, y-4)}	y > 0 /\ 2*y == y - 4
7	{(z, 2*y), (x, y-4)}	false

Path condition computation

- We traverse forward along the sequence of statements in the given path, starting with a null formula and gradually building it up. At any point during the traversal of the trace, we maintain a set of symbolic expressions for the program variables and the path condition.
  - for every assignment encountered, we update the symbolic assignment store.
  - for every branch statement encountered, we conjoin the branch condition with the path condition. While doing so, we use the symbolic assignment store for every variable appearing in the branch condition.
- At the end of the trace, we get the path condition.

Topics Covered

- Dynamic checking of programs
  - Dynamic slicing - what was important & executed
  - Hierarchical slicing – managing dynamic slices
  - Fault Localization – Trace comparison
  - Directed testing – Symbolic execution along traces
- Static checking of programs – Not covered in this module