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.
- DO-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.

The art of debugging

"A software bug (or just "bug") is an error, flaw, mistake, or failure 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
We should automatically produce the bug report via analysis of program and/or execution trace. Bug report is a small fragment of the program.

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. ⇒ 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
  - 65,000 bugs
  - 13 * $5000000 / 1000 = 65,000 bugs
  - 65,000 bugs / 44 weeks = 1477 weeks = 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

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

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);
    }
3 }
4 }
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;
- \( U := V \)
- A[i] := 1;
- \( U := A[i] \)

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

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;
4. } else{
5.     savedValue = "";
6. this.runningVersion = runningVersion;
7. }
8. System.out.println(savedValue);
Jslice: a dynamic slicing tool

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
- Dynamic Slice is still too large …
  - … for human comprehension
  - Now

An example

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

init( .. db ) {
    db = ...
}
operator ( .. db ) {
    db = ...
}
output( db ) {
    print(db...);
}
```

Divide trace into phases

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

main()
init()
operate()
output()
```
Report inter-phase dependencies

Intra-phase control and data dependencies are suppressed. Inter-phase dep. form input-output relationships.

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.

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

Slicing Criterion

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

p and q point to the same object?

- Static points-to analysis is always conservative

Relevant Slicing

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

Slicing Criterion

```
1 b=10;
2 x=1;
3 if (a>1){
 4   if (b>1){
 5     x=2;
 6   }
7   }
8 printf(“%d”, x);
```

Source of Failure

- Dynamic Slice
  - input: a=2
  - execution is omitted

- Execution is omitted
Potential Dependence

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

Relevant Slice

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

Program Slice

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
<th>Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

```
input: a=2
```

```c
Potential Dependence
```

```
Dynamic Data Dependence
```

```
Potential Dependence
```

```
Relevant Slice
```

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

- Failing Run
- Successful Run
- Compare Execution
- Difference
- As bug report
- Developer
Comparing executions

1. \( m = \ldots \)
2. if \( (m \geq 0) \) {
3. \( \ldots \)
4. lastm = m;
5. } should be
6. \( \ldots \) if \( ((m \geq 0) \&\& (lastm=m)) \)

Fault localization

Example program

Program

Example program

Set of statements

\[ S = \text{Set of statements executed in } \pi \]
\[ \{1,3,5,6,7,10\} \]
\[ S1 = \text{Set of statements executed in } \pi1 \]
\[ \{1,3,4,5,6,9,10\} \]
If \( \pi \) is faulty and \( \pi1 \) is OK
- Bug report = \( S - S1 = \{4,7\} \)
- Choice of the execution run to compare with is very important.
Another difference metric

Failing Run | Successful Runs
---|---
\( \pi \) | \( \pi_1, \pi_2 \)
\( \text{diff}_1 \) | \( \text{diff}_2 \)

\* Number of Branches
\* Location of Branches

Compare

Diff. b/w traces shown

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

Trace alignment and differences

<table>
<thead>
<tr>
<th>Execution Run</th>
<th>Alignment</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (a){</td>
<td>}</td>
<td>while (a){</td>
</tr>
<tr>
<td>1. if (b)</td>
<td>2. if (b)</td>
<td>2. if (b)</td>
</tr>
</tbody>
</table>
| \( i++ \) | \( i++ \) | \( i++ \)
| } | } | ]
| 1. while (a){ | 1. while (a){ | 1. while (a){ |
| 2. if (b) | 2. if (b) | 2. if (b) |
| \( i++ \) | \( i++ \) | \( i++ \)
| } | } | ]

3rd Loop Iteration

Compare Corresponding Statement Instances

1. while (a){
2. if (b)
3. \( i++ \)
4. }
5. ...... 1. while (a){
2. if (b)
3. \( i++ \)
4. }
5. 

Use control dependencies!

Comparison of differences

1. \( \pi \)
2. \( \pi_1 \)
3. \( \pi_2 \)
4. \( \text{diff} \)
5. \( \text{diff}^* \)
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

We should have (output > input)

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
\( Y = Y + 1 \)

\( (X = 1, Y = 1, Z = 2, W = 1) \)

\( X = Y \land Z > W \)

false

\( X = X - 1 \)

Edge coverage

Make the branch condition true/false

\( Y = Y + 1 \)

\( (X = 1, Y = 1, Z = 2, W = 1) \)

\( (X = 1, Y = 1, Z = 2, W = 2) \)

\( X = Y \land Z > W \)

false

\( X = X - 1 \)

Condition coverage

- For each executable condition \( c \)
  - Check whether it can be both true or false
  - \( c \) could be unsatisfiable or valid in all program 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

\( (X = 1, Y = 1, Z = 2, W = 1) \)

\( (X = 1, Y = 1, Z = 2, W = 2) \)

\( X \equiv Y \) is true in both the test cases

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

```c
if (Climb)
    separation = Up;
else
    separation = Up + 100;
if (separation > 150) (Climb == 0, Up == 457)
    upward = 1;
else
    upward = 0;
if (upward >0) printf("Upward");
else
    printf("Downward");
```

Generating new tests

- The path condition calculated
  - Climb ==0 ∧ Up + 100 > 150 ∧ upward > 0
- Minimally modify the condition
  - Climb ==0 ∧ Up + 100 > 150 ∧ ¬(upward > 0)
- Corresponding to the path ...

Infeasible path!!

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

Generating new tests

- The path condition calculated
  - Climb ==0 ∧ Up + 100 > 150 ∧ upward > 0
- Minimally modify the condition
  - Climb ==0 ∧ Up + 100 > 150 ∧ ¬(upward > 0)
- Corresponding to infeasible path!
- Modify a bit more
  - Climb == 0 ∧ ¬(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");

Generating new tests

- The path condition calculated
  - Climb = 0 ∧ Up + 100 > 150 ∧ upward > 0

- Minimally modify the condition
  - Climb = 0 ∧ Up + 100 > 150 ∧ ¬(upward > 0)

- Corresponding to infeasible path!

- Modify a bit more
  - Climb = 0 ∧ (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");
}