Debugging of Evolving Programs

Abhik Roychoudhury
National University of Singapore
http://www.comp.nus.edu.sg/~abhik

12/1/2010

Evolving Programs

- Code Evolution
  - Consider a banking system
    - Features: Login, Logout, View Balance, ...
    - Version 1, P1
  - Customer wants new feature, produce new version P2
    - New feature: Funds transfer
    - This breaks the functionality of "View Balance"
      - No longer see the latest balance correctly!
    - Example of regression due to code evolution
      - Different from "requirements evolution" – intended meaning of "view balance" is unchanged from P1 to P2.

12/1/2010

Problem Statement

Test Input t

Old Stable Program P → Pass
New Buggy Program P' → Fail

why?

Change Analysis?

Search among subsets!

if (x > 0) {
  y = x + 1;
  z = x;
  w = x + 2;
} else {
  y = x;
}

Requires defining the set of all changes.

Question: What if the two programs are completely different implementations of the same protocol?

12/1/2010

Trace Comparison?

Compare failing test with a similar, successful test.

Requirement: How do we find such an execution?

Question: why ignore the evolution?

12/1/2010

Adapting Trace Comparison

Test Input t

Old Stable Program P

New Buggy Program P'

Directly Compare σ and π

Path α for t

Test Input t'

New Input t'

x

Path α for t

12/1/2010
How to obtain the new test?

- We have:
  - Two versions of the program, \(P\) and \(P'\).
  - A test \(t\) that fails on \(P'\) but passes on \(P\).
- Key requirement: Similarity
  - Test \(t\) and \(t'\) are similar if they induce
    - same control flow path in \(P\) but
    - different paths in \(P'\).

How to obtain the new test?

- We have:
  - Two versions of the program, \(P\) and \(P'\).
  - A test \(t\) that fails on \(P'\) but passes on \(P\).
- Key requirement: Similarity
  - Test \(t\) and \(t'\) are similar if they induce
    - same control flow path in \(P\) but
    - different paths in \(P'\).

Our Approach

1. Solve \(f \land \neg f'\) to get another input \(t'\)
2. Compare \(\pi\) and \(\pi'\) to get bug report.

Path conditions

- Quantifier free first order logic formula
- Obtained from a path
  - \(\text{input } x, y\)
  - if \(x > 1\) \(\quad x > 1\)
  - if \(y < 20\) \(\quad x > 1 \land y < 20\)
  - ....
- Conjunction of primitive constraints with program variables.
  - All program variables implicitly existentially quantified.

Exercise

- Given a program and an input, develop an automated method to compute path conditions.
  - \(\text{input } x, y\)
  - if \(x > 1\) \(\quad x > 1\)
  - \(z = x + y\)
  - if \(z < 20\) \(\quad x > 1 \land x + y < 20\)
  - ....

Generating New Input

1. Compute \(f\), the path condition of \(t\) in \(P\).
2. Compute \(f'\), the path condition of \(t\) in \(P'\).
3. Solve for \(f \land \neg f'\)
   - Many solutions: Compare the trace of each \(t'\) in \(P'\) with the trace of \(t\) in \(P\).
   - No solution: go to next step.
4. Solve for \(f' \land \neg f\)
   - Many solutions: Compare the trace of each \(t'\) in \(P'\) with the trace of \(t\) in \(P\).
   - No solution: Impossible, since then \(f \iff f'\)
Simple Example

```c
int inp, outp;
scanf("%d", &inp);
if (inp >= 1)
    scanf("%d", &inp);
outp = g1(inp);
printf("%d", outp);
```

```c
int inp, outp;
scanf("%d", &inp);
if (inp > 9)
    outp = g(inp);
else
    outp = h(inp);
printf("%d", outp);
```

Overview of our Solution

1. Compute $f$, the path condition of $t$ in $P$.
2. Compute $f'$, the path condition of $t$ in $P'$.
3. Solve for $f \land \neg f'$.
   - Many solutions: Compare the trace of each $t'$ in $P'$ with the trace of $t$ in $P$. Return bug report from $P'$.
   - No solution: go to next step.
4. Solve for $f' \land \neg f$.
   - Many solutions: Compare the trace of each $t'$ in $P$ with the trace of $t$ in $P$. Return bug report from $P$.
   - No solution: Impossible, since then $(f \equiv f')$
Putting Everything Together

STP solver and input validation
Concrete and Symbolic Execution
Satisfiable subformulas from \( f \land \neg f' \)

Can find other potential bugs

LibPNG v1.0.7 length =
\[
\text{png_get_uint_32(chunk_length)};
\]
LibPNG v1.2.21 length =
\[
\text{png_get_uint_31(chunk_length)};
\]
\[
\text{png_get_uint_31(png_structp png_ptr,}
\]
\[
\text{png_bytep buf)}
\]
\[
\text{if (i > PNG_UINT_31_MAX)}
\]
\[
\text{png_error(png_ptr, "PNG unsigned integer out of range."));}
\]
\[
\text{return (i)};
\]

An experiment

Validate Embedded Linux
AGAINST
Linux (GNU Core-utils, net-tools)

Busybox distribution is 121 KLOC.
Various errors to be root-caused in tr, arp, top, printf.

Trying on Embedded Linux

The concept
– Golden: GNU Coreutils, net-tools
– Buggy: Busybox
  • De-facto distribution for embedded devices.
  • Aims for low code size
  • Less checks and more errors.
  • Try DARWIN!

The practice
– Failing input takes logically equivalent paths in Busybox and Core-utils.
Going beyond

Going beyond

What went wrong?

• The effect of the bug is not observable in terms of change in program paths.

A more direct approach

• Characterize observable error (obs)
  • y != 0
  • Weakest pre-condition along failing path w.r.t. obs
  • 2x != 0
  • 2x + 1 != 0
  • Compare the WPs and find differing constraints.
  • Map differing constraints to the lines contributing them.

Weakest pre-condition

WP along a path

• Along a path
  1. Start with a primitive constraint
  2. Proceed along the path from the end.
  3. For every assignment, replace occurrences of the lhs by rhs in the existing formula.
  4. Stop when you reach the beginning of the path.
  5. What kind of a constraint will you end up with?

Entire failing trace is not needed

• Along a path
  1. Start with a primitive constraint
  2. Proceed along the path from the end.
  3. For every assignment, replace occurrences of the lhs by rhs in the existing formula.
  4. Stop when you reach the beginning of the path.
  5. What kind of a constraint will you end up with?
What is the issue?

- Inherent parallelism exists in sequential programs
  - inp1 helps compute x
  - inp2 helps compute y

- Exploit the inherent parallelism to project the "relevant" part of the trace.
  - Dynamic slicing (from last lecture!)
  - Symbolic execution (WP computation) along the dynamic slice only.

- Crucial for scalability of our method!

Approach 2 - in action (simplified)

```plaintext
1. ... // input inp1, inp2
2. if (inp1 > 0)
3. x = f1(inp1);    // bug
4. else x = g1(inp1);
5. if (inp2 > 0)
6. y = f2(inp2)
7. else y = g2(inp2);
8. ... // output x, y
```

Comparing WP, WP'

- WP = (φ_1 ∧ φ_2 ∧ ... ∧ φ_n)
- WP' = (φ'_1 ∧ φ'_2 ∧ ... ∧ φ'_m)

- Check
  - WP ⇒ φ'_1 ...
  - WP' ⇒ φ_1 ...
  - Solver may choke!

- Instead, we can perform pair-wise comparison
  - Too costly ??

So, what do we do then?

- WP = Conjunction of n constraints
  - Remove tautologies
  - WP = (φ_1 ∧ φ_2 ∧ ... ∧ φ_n) x < n
- WP' = Conjunction of m constraints
  - Remove tautologies
  - WP' = (φ'_1 ∧ φ'_2 ∧ ... ∧ φ'_m) y < m

- For each φ'_j check if there is a φ_j s.t. φ_j ⇒ φ'_j
- For each φ_j check if there is a φ'_i s.t. φ'_i ⇒ φ_i
Summarizing

• Debugging evolving programs (code evolution)
  – Program Versions
  – Embedded SW against non-embedded version
  – Two implementations of same specification
    - Web-servers implementing http protocol

• Use of formal techniques into debugging
  – Beyond a “black art”.

For more ...


Also see: