Testing

Simple program – what do we want to know about it?

Not a new question for us – let's consider it with white-box testing.

```c
if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert(false)
}
```

Visser, Pasareanu & Mehlitz

---

Control flow graph (CFG)

- $x > y$
- $x = x + y$
- $y = x - y$
- $x = x - y$
- $x > y$
- assert(false)
- end

Can this statement ever be executed?

Edge coverage

- $[x=1; y=0]$
- $x > y$
- $x = x + y$
- $y = x - y$
- $x = x - y$
- $x > y$
- assert(false)
- end

- $[x=1; y=1]$
- $[x=0; y=1]$

Edge ever taken?

Symbolic execution $[x=\alpha; y=\beta]$

- $x = x + y$
- $y = x - y$
- $x = x - y$
- $x > y$
- assert(false)
- end

- $[x=\alpha+\beta; y=\beta]$

Symbolic execution

- $[\alpha > \beta]$
- $[x=\alpha+\beta; y=\beta]$
- $x = x + y$
- $y = x - y$
- $x = x - y$
- $x > y$
- assert(false)
- end

- $[x=\alpha+\beta; y=\alpha]$
- $[x=\beta; y=\alpha]$
- $[x=\beta; y=\alpha]$

$\beta > \alpha$ ever here?
What’s really going on?

- Create a symbolic execution tree
- Explicitly track path conditions
- Solve path conditions — “how do you get to this point in the execution tree?” — to define test inputs
- Goal: define test inputs that reach all reachable statements

Another example (Sen and Agha)

```c
int double (int v){
    return 2*v;
}
void testme (int x, int y){
    x = double (y);
    if (x == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Way cool — we’re done!

- First example can’t reach `assert(false)`, and it’s easy to reach `end` via both possible paths
- Second example: can reach `error` and `end` via both possible paths
- Well, what if we can’t solve the path conditions?
  - Some arithmetic, some recursion, some loops, some pointer expressions, etc.
  - We’ll see an example
- What if we want specific test cases?

Concolic testing: Sen et al.

- Basically, combine concrete and symbolic execution
- More precisely...
  - Generate a random concrete input
  - Execute the program on that input both concretely and symbolically simultaneously
  - Follow the concrete execution and maintain the path conditions along with the corresponding symbolic execution
  - Use the path conditions collected by this guided process to constrain the generation of inputs for the next iteration
  - Repeat until test inputs are produced to exercise all feasible paths

Error: possible by solving equations

```
[2 * β = α & α > β + 10]
= [2 * β > β + 10]
= [β > 10]
= [β > 10 & 2 * β = α]
```

2nd example redux

1st iteration x=22, y=7

```
int double (int v){
    return 2*v;
}
void testme (int x, int y){
    x = double (y);
    if (x == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Now solve

```
2 * β = α to force the other branch
```

```
x = 1; y = 2 is one solution
```
<2nd example

2nd iteration x=1, y=2

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
</tr>
</tbody>
</table>

int double (int v) { return 2*v; }
void testme (int x, int y) {
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }
  }
}

Now solve
\[ 2 \beta = \alpha \land \alpha <= \beta + 10 \] to force the other branch
x = 30; y = 15 is one solution

Three concrete test cases

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
| 30 | 15 | Takes first and second then

Concolic testing example

void test_me(int x,int y) {
  z = x^2 + 3x^2 + 9;
  if(x != y) {
    printf("Good branch");
  } else {
    printf("Bad branch");
    abort();
  }
}

Random seed
- \[ x = -3; y = 7 \]
Concrete
- \[ z = 9 \]
Symbolic
- \[ z = x^2 + 3x^2 + 9 \]

Take then branch with constraint
\[ x^2 + 3x^2 + 9 = y \]

Concolic testing example

void test_me(int x,int y) {
  z = x^2 + 3x^2 + 9;
  if(x != y) {
    printf("Good branch");
  } else {
    printf("Bad branch");
    abort();
  }
}

Random memory graph reachable from \( p \)
Random value for \( x \)
Probability of reaching \( \text{abort}() \) is extremely low

Typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0) {
    if (p != NULL) {
      if (f(x) == p->v) {
        if (p->next == p) {
          abort();
        }
      }
    }
  }
  return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
    int f(int x) {
        return 2*v + 1;
    }
    int testme(cell *p, int x) {
        if (x > 0) {
            if (p != NULL) {
                if (f(x) == p-v) {
                    if (p->next == p) abort();
                }
                return 0;
            }
        }
    }
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0) {
        if (p != NULL) {
            if (f(x) == p-v) {
                if (p->next == p) abort();
            }
            return 0;
        }
    }
}

p=NULL;
x=236

Concolic: status

- The jury is still out on concolic testing — but it surely has potential
- There are many papers on the general topic
- Here's one that is somewhat high-level Microsoft-oriented
  - They tend to call the approach DART — Dynamic Automated Random Testing

Next steps

- Worksheets