Test Data Generation

- Introduction
- Symbolic Execution
- Search-Based Generation
- Summary

The Grand Challenge

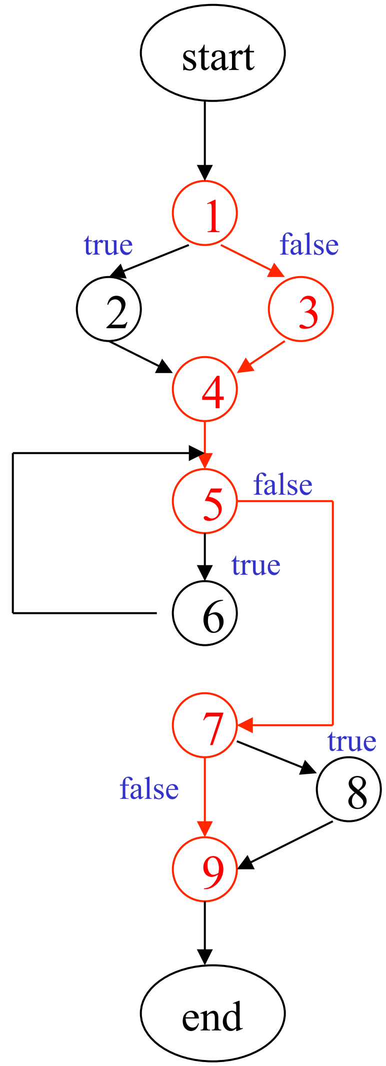
- Given a test path, how to generate data input such that the path is executed?

Main Idea

- Collect the branching conditions into a path condition
  - Symbolic execution is used to rewrite internal variables as external inputs
- Find data input that satisfy the path condition
  - Use a constraint solver to solve the path condition
  - Search the input space randomly or in a more systematic manner
- A simple idea but very challenging to implement

Example

```
begin
    int x, y, power;
    float z;
    input (x, y);
    if (y < 0)
        power = -y;
    else
        power = y;
    z = 1;
    while (power != 0) {
        z = z * x;
        power = power – 1;
    }
    if (y < 0)
        z = 1/z;
    output (z);
end
```



PC: ! (y < 0)
&& ! (power !=0)
&& ! (y < 0)

PC: ! (y < 0)
&& ! (y !=0)
&& ! (y < 0)

PC: (y = 0)

Major Challenges

- Pointer: Which object does a pointer actually point to?
- Array: Which element does an array index variable actually refer to?
- Complexity of constraint handling, especially for non-linear constraints
- Many others such as database, concurrency, native code.

Arrays and Pointers

```
a[i] = 0;
a[j] = 1;
if (a[i] > 0) {
    // perform some actions
}
```

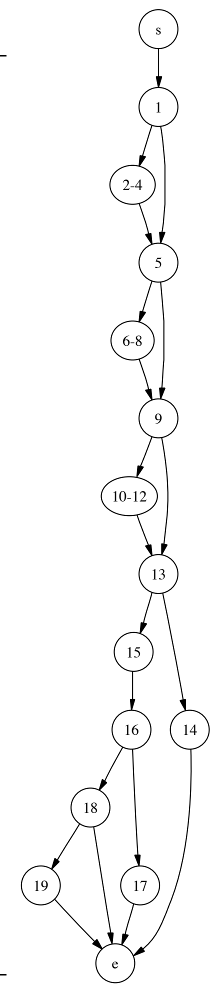
```
*a = 0;
*b = 1;
c = *a;
```

Symbolic Execution

- A technique that executes a program taking symbols as inputs, instead of concrete values
- Mainly used to derive a path condition in terms of the input variables
- A process that essentially collects branching conditions and rewrites internal variables in terms of external inputs

Example

CFG Node	
s	`int tri_type(int a, int b, int c)`
	`{`
	`    int type;`
1	`    if (a > b)`
2-4	`    {   int t = a; a = b; b = t;   }`
5	`    if (a > c)`
6-8	`    {   int t = a; a = c; c = t;   }`
9	`    if (b > c)`
10-12	`    {   int t = b; b = c; c = t;   }`
13	`    if (a + b <= c)`
	`    {`
14	`        type = NOT_A_TRIANGLE;`
	`    }`
	`    else`
	`    {`
15	`        type = SCALENE;`
16	`        if (a == b && b == c)`
	`        {`
17	`            type = EQUILATERAL;`
	`        }`
18	`        else if (a == b || b == c)`
	`        {`
19	`            type = ISOSCELES;`
	`        }`
	`    }`
e	`    return type;`
	`}`



Example

- Assume that the following path is to be executed: <s, 1, 5, 9, 10, 11, 12, 13, 14, e>
- Three symbols are assigned to the input variables: a = i, b = j, c = k.

Example (2)

- Node 1, false: i <= j
- Node 5, false: i <= k
- Node 9, true: j > k
- Node 10 - 12: t = j, b = k, c = t
- Node 13, true: i + k <= j
- PC: (i <= j && i <= k && j > k && i + k <= j)
- One possible solution: i = 10, j = 40; k = 20

If nodes 10-12 are not executed, i.e., b and c do not exchange their symbols, then at node 13, we would derive condition, i + j <= k, which would not be correct.

Random Testing

- Simply try random inputs and observe the program execution until the path of interest is executed.
- Assume that each side can take a value from 1 to 100. What is the probability for the three sides to be the same value?

Local Search

- Consider the same path: <s, 1, 5, 9, 11, 12, 13, 14, e>
- Let (a = 10, b = 20, c = 30) be the initial program input, which diverges at node 9.
- At this point, a local search can be used to change program inputs so that the alternative branch is taken.

Alternating Variable Method

- Each variable is taken in turn and its value is changed whereas other variables are not changed.
- Exploratory Phase: Explore the neighborhood of the variable to determine the direction of improvement
- Pattern Phase: Make a series of moves to reach the target

Example

- Consider the digression at node 9.
- Change a has does not help.
- Increase of b leads to improvement.
- Increase b’s value until b > c. Assume that b is 31.
- Now the execution proceeds as desired, and repeat the same process on node 13.
- Eventually we found (a = 10, b = 40, c =30) that executes the entire path.

A Goal-Oriented Approach

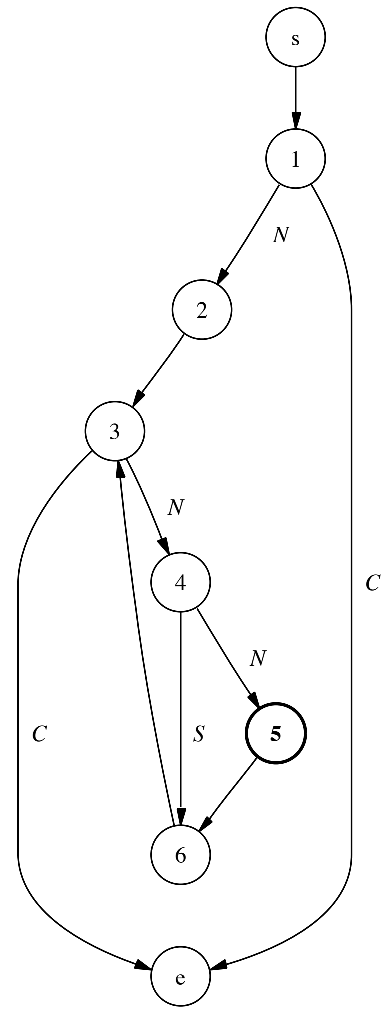
- Earlier approaches are path-oriented in that a specific path is given to be covered.
- A goal-oriented approach tries to automatically select a path to reach a particular statement.

Types of Branches

- Critical branch: a branch that leads the execution away from the target node
  - The alternating variable method is used to find inputs so that the other branch is taken
- Semi-critical branch: a branch that leads to the target node, but only via the backward edge of a loop.
- Non-essential branch: a branch that is neither critical or semi-critical branch.

An Example Program

CFG Node	
s	`void goal_oriented_example(int a)`
	`{`
1	`if (a > 0)`
	`{`
2	`int b = 10;`
3	`while (b > 0)`
	`{`
4	`if (b == 1)`
	`{`
5	`// target`
	`}`
6	`b --;`
	`}`
	`}`
e	`return;`
	`}`



Example (3)

- Assume that the input vector is (a = 0).
- Node 1: The value of a is changed to take the true branch
- Node 4: The semi-critical branch is allowed to be taken
- The loop is executed for 9 more times until the true branch of node 4 is taken, which allows the target node to be reached.

Potential Problems

```
void nested_example(int a, int b, int c)
{
    if (a == b)
        if (b == c)
            if (c < 0)
                // target
}
```

Assume that the initial input vector is (a = 10, b = 10, c = 10).

Potential Problems (2)

CFG Node	
s	`void chaining_approach_example(int a)`
	`{`
1	`    int b = 0;`
2	`    if (a > 0)`
	`    {`
3	`        b = a;`
	`    }`
4	`    if (b >= 10)`
	`    {`
5	`        // target`
	`    }`
	`    // ...`
	`}`

Potential Problems (3)

- Assume that the initial value of a is less than 0.
- The search fails because small exploratory moves have no effect on the value of b.
- Possible solutions?

Summary

- Test data generation is one of the most challenging problems in software testing.
- Symbolic execution allows us to derive the path condition which if satisfied, causes a path to be executed.
- Constraint solving and search-based methods are two general approaches to find program inputs that satisfy a given path condition.
- Significant challenges remain in terms of how to handle complex control and data structures.

Reference

P. McMinn, Search-Based Software Test Data Generation: A Survey, Software Testing, Verification and Reliability, 14(2), pp. 105-156, June 2004.