Test Data Generation

Software Testing

Instructor : Ali Sharifara
Quick Review

• What is regression Testing?
• What is the main idea of test minimization?
• What is the purpose of test prioritization and why we need to prioritize the tests?
Test Data Generation

- Basic Concepts
- Path-oriented data generation
- Goal-oriented data generation
- Types of Branches
- Summary
Test Data Generation

**Given**: a target

**Goal**: find a program input on which the target is executed
Test Data Generation Methods

- Random test generation
- Path-oriented test generation
- Symbolic execution oriented test generation
- Execution-oriented test generation
- Goal-oriented test generation
- …
F(int a[10], int b[10], int target) {
    int i;
    bool fa, fb;
    i=1;
    fa=false;
    fb=false;
    while (i < 10 {
        if (a[i] == target) fa=true;
        i=i + 1;
    }
    if (fa == true) {
        i=1;
        fb=true;
        while (i < 10) {
            if (b[i] != target) fb=false;
            i=i+1;
        }
    }
    if (fa == true) {
        i=1;
        fb=true;
        while (i < 10) {
            if (b[i] != target) fb=false;
            i=i+1;
        }
    }
    if (fb==true) printf("message1");
    else printf("message2");
}
• A statement
• A branch
• A path
• A data flow
• A multiple condition
• An assertion
• A specific output value
• …
Test Data Generation

- Basic Concepts
- Path-oriented data generation
- Goal-oriented data generation
- Types of Branches
- Summary
Target statement $S$

Select path $P$ to target statement $S$

Find input to execute path $P$

Input found? yes

An input to execute path $P$ and target statement $S$
Example of Path-Oriented Test Generation

1. input (a,n);
2. max=a[1];
3. min=a[1];
4. i=2;
5. while (i<=n) {
6,7  if (max<a[i]) max=a[i];
8,9  if (min>a[i]) min=a[i];
10  i=i+1;
}
11. output(min,max);
The Grand Challenge

- Given **one or more test paths**, how to generate **data inputs** such that these paths are **executed**?
- It is not easy to **anticipate** the **paths** that the program could take making it nearly impossible for the test data generators to generate comprehensive test data.
- Selected paths are frequently non-executable.
- A lot of search effort is “wasted” on non-executable paths.
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
Major Challenges

• **Loop**: How many times do we need to execute a loop?
• **Pointer**: Which object does a pointer actually point to (dynamic allocation of pointers)?
• **Array**: Which object does an array variable actually refer to (dynamic allocation of arrays)?
• **Objects**: determine which code will be called at runtime
• **Complexity of constraint handling**, especially for non-linear constraints
• **Many others such as database, concurrency, native code.**
Arrays and Pointers

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

The evaluation of the predicate depends on the values of i and j which are not known at compile time. Similarly it is not known whether a and b refer to the same object.

We do not know whether i = j, which affects the evaluation of the predicate.
Some Approaches

- **Input Space Testing**: Try to cover the input space, instead of the program structure.
- **Model-Based Testing**: Create a model of the expected system behavior and then derive tests from the model.
- **Exploratory Testing**: Execute a program with an arbitrary input, record the path (in terms of branching conditions), and then change one or more branching conditions to explore different paths.
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

```plaintext
x = read();
y = 5 + x;
z = 7 + y;
a[z] = 1
```

Concrete Memory

- \(x \rightarrow 5\)
- \(y \rightarrow 10\)
- \(z \rightarrow 17\)
- \(a \rightarrow \{0,0,0,0\}\)

Symbolic Memory

- \(x \rightarrow \alpha\)
- \(y \rightarrow 5 + \alpha\)
- \(z \rightarrow 12 + \alpha\)
- \(a \rightarrow \{0,0,0,0\}\)
Path condition

- Program control can be effected by symbolic values

```plaintext
1  x = read();
2  if (x>5) {
3      y = 6;
4      if (x<10)
5          y = 5;
6  } else y = 0;
```

- We represent the influence of symbolic values on the current path using a path condition $\pi$
  - Line 3 reached when $\alpha > 5$
  - Line 5 reached when $\alpha > 5$ and $\alpha < 10$
  - Line 6 reached when $\alpha \leq 5$
Path feasibility

Whether a path is **feasible** is equivalent to a path condition being satisfiable

```
x = read();
if (x>5) {
    y = 6;
    if (x<3)
        y = 5;
} else y = 0;
```

Solution to path constrains can be used as inputs to a concrete test case that will execute that path

Solution to reach line 3: $\alpha = 6$

Solution to reach line 6: $\alpha = 2$
Symbol Execution Tree

```c
int a = α, b = β, c = γ;
    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
    x = -2;
}
if (b < 5) {
    if (!a && c) { y = 1; }
    z = 2;
}
assert(x+y+z!=3)
```
Applications of Symbolic Execution

- **General goal**: identifying semantics of programs

- **Basic applications**:
  - Detecting *infeasible paths*
  - Generating *test inputs*
  - Finding *bugs* and *vulnerabilities*
  - Proving two code segments are equivalent

- **Advanced applications**:
  - Generating program invariants
  - Debugging
  - Repair programs
Detecting Infeasible Paths

Suppose we require $\alpha = \beta$

```
int a = a, b = b, c = c;
    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
    x = -2;
} if (b < 5) {
    if (!a && c) { y = 1; }
    z = 2;
} assert(x+y+z!=3)
```
Random Test Data Generators

- Simply try random inputs and observe the program execution until the path of interest is executed.
  - Program is executed several times
  - On every execution: check, whether or not the desired path is executed
  - If desired path is not executed, program is re-executed with slightly modified input values
  - Program is re-executed until desired path is traversed or user-defined limit (time, execution count) is exceeded
  - Solves some problems of symbolic execution since values of variables are available
Test Data Generation

- Basic Concepts
- Path-oriented data generation
- **Goal-oriented data generation**
- Types of Branches
- Summary
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.

The approach starts by executing the program on any input. When the program is executed, the program execution flow is monitored. During the execution, the search procedure decides whether the execution should continue through the current branch or should take an alternative branch, because the current branch does not lead to the target.
Goal-Oriented Test Generation

execute program on any input

this execution does not lead to the target

problem node

this execution may lead to the target

target statement
Goal-Oriented Test Generation

1. input (a,n);
2. \( \text{max} = a[1] \);
3. \( \text{min} = a[1] \);
4. \( i = 2 \);
5. while (\( i \leq n \)) {
6. \quad \text{if} (\text{max} < a[i]) \text{ max} = a[i];
7. \quad \text{if} (\text{min} > a[i]) \text{ min} = a[i];
8. \quad i = i + 1;
9. }
10. output(\text{min}, \text{max});
Goal-Oriented Test Generation

1. input (a,n);
2. max=a[1];
3. min=a[1];
4. i=2;
5. while (i<=n) {
   6,7 if (max<a[i]) max=a[i];
   8,9 if (min>a[i]) min=a[i];
   10 i=i+1;
}
11. output(min,max);

Initial input:

a={2, 7}, n = -5
Goal-Oriented Test Generation

1. input (a,n);
2. max=a[1];
3. min=a[1];
4. i=2;
5. while (i<=n) {
   6, 7   if (max<a[i]) max=a[i];
   8, 9   if (min>a[i]) min=a[i];
   10   i=i+1;
}
11. output(min,max);

Initial input:
   a={2, 7}, n= -5

F=i-n=7

find new value of a and n such that F<=0
Goal-Oriented Test Generation

• There are many searching algorithms that can be used to find a new program input based on the fitness function
  • Hill-climbing algorithm
  • Simulated annealing
  • Evolutionary algorithm
  • …
Test Data Generation

- Basic Concepts
- Path-oriented data generation
- Goal-oriented data generation
- Types of Branches
- Summary
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

```c
void goal_oriented_example(int a)
{
    if (a > 0)
    {
        int b = 10;
        while (b > 0)
        {
            if (b == 1)
            {
                // target
            }
            b --;
        }
    }
    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.
Test Data Generation

- Basic Concepts
- Path-oriented data generation
- Goal-oriented data generation
- Types of Branches
- Summary
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.

• **Random Test Data Generators** simply try random inputs and observe the program execution until the path of interest is executed.

• In **Goal oriented** approach, there are many searching algorithms that can be used to find a new program input based on the fitness function.

• Significant challenges remain in terms of how to handle complex control and data structures.