Software Verification and Testing

Lecture Notes: Testing I
Motivation

verification:

• powerful method for finding software errors
• mathematical proof of absence of errors in implementations relative to specifications
• formal specification and analysis often very expensive; requires highly qualified engineers
• automated techniques rather limited

testing: (as “poor man’s verification”)

• can only detect presence of errors
• cannot find all errors (induction problem)
• much cheaper than verification
• requires less mathematical skills
Motivation

verification vs testing:

- verification used often in early stages of development; testing in later stages
- but test cases can be developed during the specification phase
- verification presupposes formal program semantics; testing does not
- verification often based on abstraction, thus also only a necessary correctness criterion
- system tests go beyond verification, since real environment is involved
- testing is strongly used in software engineering: up to 40% of software development efforts go into it
- formal verification is rarely used in practice...
Motivation

psychology of testing:
- coding is often seen as a “constructive” or “creative” activity; testing as a “destructive” one
- the aim of testing should not be in the verification, but in the falsification of a program
- ideally, development and testing teams should be separate

reality of testing:
- testing can be partially automated (using CASE-tools)
- good testing may require considerable engineering experience
- testing strongly contributes to software quality
Testing Notation

**test object:** the software component or program to be tested

**test case:** a collection of test data causing the complete execution of the test object

**test datum:** input value for an input variable or input parameter of the test object in a particular test case

**test driver:** frame for interactively calling a test object which is a function or procedure
Testing Notation

**instrumentation:**
- addition of counters to source code
- either manually or by a tool
- evaluation of counters gives information about commands executed

**coverage:** describes the degree of completeness of a test procedure

**regression tests:** automated replay of test cases after code alternations
Classification of Testing Methods

dynamic testing: software component is executed with concrete input values (in a real environment)

- structure testing (white-box testing):
  test cases derived from control flow or data flow of the component
- functional testing (black-box testing):
  test cases derived from (formal) component specification

static testing: code analysis (components are not executed) by
  code inspections, code reviews, walkthroughs.

symbolic execution: (abstract interpretation) execution of source code with
  abstract symbolic input values by interpreter;
  intermediate between testing and verification
Classification of Testing Methods

here: focus on dynamic testing

structure testing:

• control flow oriented:
  statement coverage tests, branch coverage tests, path coverage tests,
  condition coverage tests
• data flow oriented:
  defs/uses-tests, required k-tuples tests

functional testing: equivalence class tests, boundary value tests,
  special value tests, random value tests, state automata tests
example: specification

- the procedure reads input from the keyboard; it stops when some input is not an upper case character or some upper value MaxSize has been reached
- if the input an upper case character, then the counter InCount is incremented; if it is a vowel, then the counter VoCount is incremented
- both counters are input and output parameters
- the invariant VoCount <= InCount holds
Control Flow Graphs

example: implementation

```cpp
void CountDigits(int &VoCount, int &InCount)
{
    char Digit
    cin >> Digit       //read Digit from input stream
    while ((Digit >= 'A') && (Digit <= 'Z') && (InCount < MaxSize))
    {
        InCount++;
        if ((Digit == 'A') || (Digit == 'E') || (Digit == 'I') ||
            (Digit == 'O') || (Digit == 'U'))
        {
            VoCount++;
        } // end if
        cin >> Digit;
    } // end while
}
```
Control Flow Graphs

control flow graph: directed graph (transition system) with start and end vertex
- nodes labelled by executable commands
- edges represent control flow between nodes

basic flow graphs:

sequence

conditional

while
Control Flow Graphs

element:

```
cin >> Digit; InCount++; if ((...)) cin >> Digit; VoCount++; while ((...))
```

```
Control Flow Oriented Structure Testing

here: we consider

- **statement coverage test:**
  test case must cover all nodes, i.e., all possible commands must be executed at least once

- **branch coverage test:**
  test case must cover all edges, i.e., all possible choices must be explored at least once

- **path coverage test:**
  test case must cover all different traces of a program, i.e., paths in the flow graph

- **condition coverage test:**
  test case for complex conditions/tests
Statement Coverage

example:

```
cin >> Digit;
InCount++;
if ((...)) cin >> Digit;
VoCount++;
while ((...))
```

```
cin >> Digit;
while ((...))
InCount++;
if ((...))
VoCount++;
```

```
cin >> Digit;
```
Statement Coverage

test case: call CountDigits with InCount = 0 = VoCount

- input from keyboard: 'A', '1'
- test path: $n_i, n_1, n_2, n_3, n_4, n_5, n_6, n_2, n_o$

remark: edge from $n_4$ to $n_6$ is not considered

evaluation:

- non-executable code can be found
- not a stand-alone testing technique
Branch Coverage

eexample:

```
cin >> Digit;
while ((...))
InCount++;
if ((...))
VoCount++;
cin >> Digit;
```

```
Branch Coverage

test case: call CountDigits with InCount = 0 = VoCount

• input from keyboard: 'A', 'B', '1'
• test path: $n_i, n_1, n_2, n_3, n_4, n_5, n_6, n_2, n_3, n_4, n_6, n_2, n_0$

remarks: branch coverage

• subsumes statement coverage
• is a minimal testing technique
• helps to identify and optimise strongly used program parts

problems: branch coverage does not

• suffice for loop testing
• consider dependencies between branches
• resolve complex conditions/tests
Path Coverage

define the paths

\[ c_1 = (n_i, n_1) \cdot (n_1, n_2) \quad c_2 = (n_2, n_3) \cdot (n_3, n_4) \quad c_3 = (n_4, n_5) \cdot (n_5, n_6) \]
\[ c_4 = (n_5, n_6) \quad c_5 = (n_6, n_2) \quad c_6 = (n_2, n_6) \]

then the set of all paths can be described by the regular expression

\[ c_1 \cdot (c_2 \cdot (c_3 \cup c_4) \cdot c_5)^* \cdot c_6 \]

it can be obtained by unwinding the control flow graph
Path Coverage

question: how many paths are in \((c_1 \cup c_2)^*\) (when the maximal length of paths is bounded by \(n\))? 

answer: exponentially many \((2^n)!\)!

consequence: it is not feasible to test all possible execution paths of a component

heuristics:

- boundary-interior path test:
  1. consider all paths that enter, but do not repeat a loop (boundary test)
  2. consider all paths that repeat a loop, restricted to two repetitions (interior test)

- structured path test: generalisation of the above (discussion later)
Boundary-Interior Path Test

**example:** consider again CountDigits

1. outside of loop:
   call with InCount $= \text{MaxSize}$
   input from keyboard: whatever
   test path: $c_1 \cdot c_6$
Boundary-Interior Path Test

**example:** consider again CountDigits

2. boundary test:

(a) call with $\text{InCount} = 0$
   input from keyboard: 'A', '1'
   test path: $c_1 \cdot c_2 \cdot c_3 \cdot c_5 \cdot c_6$

(b) call with $\text{InCount} = 0$
   input from keyboard: 'B', '1'
   test path: $c_1 \cdot c_2 \cdot c_4 \cdot c_5 \cdot c_6$
Boundary-Interior Path Test

Example: consider again CountDigits

3. Interior test:

(a) call with InCount = 0
   input from keyboard: 'A', 'U', '1'
   test path: \( c_1 \cdot (c_2 \cdot c_3 \cdot c_5)^2 \cdot c_6 \)

(b) call with InCount = 0
   input from keyboard: 'U', 'K', '!'
   test path: \( c_1 \cdot c_2 \cdot c_3 \cdot c_5 \cdot c_2 \cdot c_4 \cdot c_5 \cdot c_6 \)
Boundary-Interior Path Test

**example:** consider again CountDigits

3. interior test:

(c) call with InCount = 0
input from keyboard: 'C', 'A' 'n'
test path: \( c_1 \cdot c_2 \cdot c_4 \cdot c_5 \cdot c_2 \cdot c_3 \cdot c_5 \cdot c_6 \)

(d) call with InCount = 0
input from keyboard: 'G', 'B' 'DD'
test path: \( c_1 \cdot (c_2 \cdot c_4 \cdot c_5)^2 \cdot c_6 \)
Structured Path Test

idea: extend boundary-interior path tests to depth $k$

properties: for some $k$

- do not explore paths $c_i \cdot c_j^{>k} \cdot c_l$
- explore all paths $c_i \cdot c_j^{<k} \cdot c_l$
Condition Coverage Test

example: CountDigits contains two conditions/tests

... 
((Digit >= 'A') && (Digit <= 'Z') && (InCount < MaxSize))
...

((Digit == 'A') || (Digit == 'E') || (Digit == 'I') ||
 (Digit == 'O') || (Digit == 'U'))
...

observation: path coverage tests do not analyse these conditions
Condition Coverage Test

variations:

- **simple condition coverage**: every atomic condition must be at least once true and once false
- **multiple condition coverage**: considers full truth table
- **minimal multiple condition coverage**: every condition (atomic or composite) must be at least once true and once false
Condition Coverage Tests

discussion:

• simple condition coverage subsumes not even statement coverage: not an applicable technique
• multiple condition coverage considers exponentially many cases; many of them not reachable because of dependencies
• minimal multiple condition coverage is more difficult to establish
Control Flow Testing: Empirical Data

**error identification:**

- statement coverage: 18%
- branch coverage: 34% (79% control flow errors, 20% computation errors)
- path testing techniques: no reliable data found
- condition coverage: no reliable data found
Data Flow Oriented Testing

**idea:** use definitions and accesses to variables for defining test cases

**applications:** this is useful for testing

- data structures
- data types
- objects

**variants:**

- defs/uses procedures
- required k-tuples testing
- data context covering
Defs/Uses Procedures

variables: they are used essentially in three different ways in programs

- assignments of values/definitions (defs)
- computations (c-uses)
- conditions/propositions (p-uses)

example: in if $z > 1$ then $y = x + 1$ else skip

- $z$ is p-used
- $y$ is defined
- $x$ is c-used
Defs/Uses Procedures

example:

```
cin >> Digit;
InCount++;
if ((...)) cin >> Digit;
VoCount++;
while ((...))
```

```
```

```c++
cin >> Digit;
while ((...))
InCount++;
if ((...))
VoCount++;
cin >> Digit;
```
Defs/Uses Procedures

example:

- $n_i$: def InCount, VoCount
- $n_1$: def Digit
- $n_2$: p-use Digit, InCount
- $n_3$: c-use InCount, def InCount
- $n_4$: p-use Digit
- $n_5$: c-use VoCount, def VoCount
- $n_6$: def Digit
- $n_o$: c-use InCount, c-use VoCount
Defs/Uses Procedures

**terminology:**

- a def of $x$ in $n_i$ precedes a c-use or p-use of $x$ in $n_j$
  if there is a path $c$ with source $n_i$ and target $n_j$ and
  $x$ is defined nowhere on $c$
- conversely, the c-use or p-used succeeds the def
- a p-use or c-use of a variable is **local** if it is preceded by a def
  in the same block
- it is called **global** if it is preceded by a def not in the same block
- a def of a variable is **local** if it precedes a p-use or c-use in the same block
- it is called **global** if it does not precede a use in the same block
Criteria

all defs:

• test case contains for every globally defined variable at some node a path to some succeeding c-use or p-use
• subsumes neither statement nor branch coverage

all p-uses:

• test case contains for every globally defined variable at some node a path to all succeeding p-uses
• subsumes branch coverage

all c-uses:

• test case contains for every globally defined variable at some node a path to all succeeding c-uses
• subsumes neither statement nor branch coverage
Criteria

all c-uses/some p-uses:
1. try all c-uses
2. if there is no c-use, test some succeeding p-use

all p-uses/some c-uses: dual to above

all uses: combine all c-uses and all p-uses

example: in exercises. . .
Data Flow Testing: Empirical Data

study:

- all defs, all p-uses and all c-uses together found 70% of program errors
- all c-uses found 48% of errors, first of all computation errors
- all p-uses found 34% of errors, first of all control flow errors
- all defs found 24% of errors, no control flow errors
Alternatives

required k-tuples:

- test alternating sequences of definitions and uses
- different bounds on sequence length yield different procedures

**data context coverage:** for each program variable, test each possible assignment of value