SE 465: Testing

Michael Noukhovitch

Winter 2015, University of Waterloo

Notes written from Patrick Lam's lectures.

Contents

1 Introduction

1.1 Types of Problems

- **fault**: static defect in the software
  - **design fault**
  - **mechanical fault**
- **error**: have incorrect state
- **failure**: external incorrect behavior

Example 1.1. Faults

```
static public int findLast (int[] x, int y) {
    for (int i=x.length-1; i>0; --i){
        if (x[i] == y){
            return i;
        }
    }
    return -1;
}
```

fault: should be `i >= 0`
no **fault** input: `x = null`
fault but not **error** input: `x[0] != y`
error but not **failure** input: `y not in x`

1.2 RIP model

RIP model: three things necessary to observe a failure

1. **Reachability**: PC must reach that point in the program
2. **Infection**: after fault, program state must be incorrect
3. **Propogation**: infected state propogates to cause bad output

1.3 Dealing with faults

We have three ways to deal with faults:

- **avoidance**: design, use better language
- **detection**: testing
- **tolerance**: redundancy, isolation

2 Testing

2.1 Testables

- code coverage
- output of a function
- logic coverage
- input space coverage

2.2 Types of testing

static testing: testing without running the code

- compilation
- semantic verification
- code reviews

dynamic testing: testing by running and observing the code

- **test cases**: single input, single output (wrt to some code)
- **black-box testing**: don't look at system implementation
- **white-box testing**: base tests on system's design

2.3 Coverage

We find a reduced space and cover that space with our tests

test requirement : a specific element (of software) that a test case must satisfy or cover

infeasible test req : impossible coverage e.g. unreachable code

subsumption : when one testing criterion is strictly more powerful than another criterion

3 Graph Coverage

test path : considering our test as some path through our program from some initial node in N_0, along different nodes that ends up at a final node in N_f

subpath : a path which is a subsequence of a path

3.1 Behaviours

- **deterministic**: 1 test path per test case
- **non-deterministic**: multiple test paths are possible

3.2 Reachability

- **syntactically**: reachable via edges and nodes
- **semantically**: there exist input that gets to a certain node

3.3 Coverage Criterion

Node Coverage : for every statement (node), there must be a test case that executes it

Edge Coverage : for every branch (edge), there must be a test case that goes through it

Edge-Pair Coverage : for every path (length <= 2), there must be a test case

3.4 Control Flow Graph

The fundamental graph for source code is the **Control Flow Graph** (CFG)

- **CFG node**: zero or more statements
- **CFG edge**: indicates that statements follow one another

Group together statements that are always consecutive into a **Basic Block**, with one entry and one exit

4 Path Coverage

4.1 Definitions

simple path : no node appears more than once in the path (first and last can be the same)

prime path : a simple path that is not a proper subpath of any other simple path, each node only appears once *except* first and last node

bridge : an edge which, when removed, results in a disconnected graph

4.2 Coverage Criterion

Complete Path Coverage : cover paths of all lengths

Prime Path Coverage : cover every prime path

Single Round Trip Coverage : at least one round trip (starts = end) path for each reachable node

Complete Round Trip Coverage : all round trip paths for each reachable node

Specified Path Coverage : specified set of paths

Bridge Coverage : cover all bridges

5 Concurrency

5.1 Races

Race two concurrent accesses to the same memory and one of them is a write

Race freedom doesn't guarantee bug freedom, need to test code extra:

- run multiple times
- add noise
- Helgrind ...
- force scheduling
- static approaches

5.2 Recursive Locks

If in one thread, there are two requests for one lock, the thread wait forever
`ReentrantLocks` know how many times they have been locked and need to be unlocked the same amount to liberate

- explicit `lock()` and `unlock()`
- `trylock()`

5.3 Bad Lock Usage

Lock and unlock must be paired, and comments must sufficiently describe conditions
Deadlocks can occur if an interrupt uses the same lock as your program.

- `spin_lock_irqsave` disables interrupts locally and provides `spin_lock` on symmetrical mulitiprocessors (**SMPs**)
- `spin_lock_irqrestore` restores interrupts to state when lock is acquired

5.4 Testing

- run mulitple times
- add noise (*e.g. sleep, wait*)
- Helgrind
- force different scheduling

6 Assertions

Assertion: statement about the program that is true

- **precondition**: reasoning about the callee
- **postcondition**: reasoning about the caller

6.1 Tools

Coverity

iComment

...

6.2 Beliefs

Sometimes we do not know the truth (expected behaviour) but we can infer beliefs about how the code works.

Must belief: things that must be true in the code, any condtradiction is error

```
x = *p / x // p is not NULL
           // z != 0
```

May belief: things that correlate with each other in code

```
A(); ... B();
A(); ... B(); // A and B may be paired
```

Check these beliefs as "must beliefs" and cross-check against must beliefs

1. record every sucessful MAY-belief as "check"
2. record every unsucessful as "error"
3. rank errors based on "check" : "error" ratio

6.3 Linters

Just because code compiles, doesn't mean it all variables are defined

```
function main(x) {
    if (x) {
        console.log('Yay');
    }
    else {
        console.log(num);
    }
}
main(true);
```

We can use **JSHint** to check that all top-level symbols resolve. On top of that we can use **pre-commit hooks** to make sure that all we checked into our repo passes the hooks. Improve further by forcing only master branch to go through all our pre-commit tests

7 Data Flow Criteria

Testing should take into account the data within nodes *du*-**pairs**: defintion-use pairs of nodes for variables

```
int x = 5; // definition
...
printf(x); // use
```

If we consider the def line to be n_0 and the use n_1, then def(n_0) = use(n_1) = x

7.1 Defintions

def-clear : if our variable v is not defined anywhere on our path

use reached : if our variable has a def-clear path from its defintion to use

du-**path** : a simple path that is def-clear wrt v from $n_i \leftarrow n_j$ where v in def(n_i) and use(n_j)

def-path set : fix a def and a variable, du(n_0, x)

def-pair set : fix a def, a use, and a variable, du(n_0, n_1, x)

7.2 Coverage

All-Defs Coverage : a TR for each def to one use

All-Uses Coverage : a TR for each def to all uses

All-DU-Paths Coverage : all paths from each def to all uses

7.3 Source Code

Def : x is assigned, defined as paramter in method, input to program

Use : x occurs in an expression that the program evaluates

Reachability : if *it is possible* that the address at the def refers to the same one as the use

7.4 Design Elements

Testing beyond single methods to "design elements" aka **integration testing**. We use **call graphs** where design elements are nodes and calls are edges

Caller : unit that invokes callee

Actual Parameter : value passed to callee

Formal Parameter : placeholder for incoming value

Last-def : the definiton that goes through to a call

First-use : the first use in a method, it picks up the last-def definition

Use-clear : a path that is clear of uses, except for at the start and end

8 Syntax-Based Testing

8.1 Input Testing

Use grammars to validate inputs in our program and generate test inputs.

8.1.1 Defintions

Start Symbol : the first symbol we use

Non-terminal : symbols that are defined by production rules

Terminal : symbols not defined by production rules

Production rule : how a symbol is defined by other symbols

Example 8.1. Grammar symbols:

```
actions = action* // actions: start symbol
action = dep|deb  // action: non-terminal
dep = 'deposit' account amount
deb = 'debit' account amount
account = digit{3} // = digit{3}: production rule
amount = '\$'digit+'.'digit{2}
digit = [0-9]     // 0-9: terminals
```

8.1.2 Coverage

Terminal Symbol Coverage : TR contains each terminal of grammar G

Production Coverage : TR contains each production of grammar G

Derivation Coverage : TR contains every possible string derivable from grammar G

8.1.3 Grammar Mutation

Use different operators to generate invalid input:

- Non-terminal Replacement
- Terminal Replacement
- Terminal and Non-terminal Deletion
- Terminal and Non-terminal Duplication

8.1.4 Fuzzing

Fuzzing: feed random valid characters in an effort to find bugs

- Generation-based: generate random inputs that match your grammar, use different levels of inputs, from random ASCII $\rightarrow$ correct C code $\rightarrow$ model-correct C
- Mutation-based: randomly modify existing inputs, making sure to keep checksums updated

8.2 Mutation Testing

Generate mutant by modifying programs and try to kill it with test cases

8.2.1 Defintions

Ground String : a valid string in the language of the grammar

Mutation Operator : a rule specifying synatactic variations of strings from the grammar

Mutant : result of one application of a mutation operator to a ground string

Uninteresting Mutants:

- **Stillborn**: does not compile or immediately crashes
- **Trivial**: killed by almost any test case
- **Equivalent**: indistinguishable from original program

8.2.2 Coverage

A test case t **kills** a mutant m if t produces a different output on m than m_0

Mutation Coverage : for each mutant m, TR required to kill m

Mutation Operator Coverage : for each mutation operator *op*, TR required to kill all mutant derived using *op*

Mutation Production Coverage : for each mutation operator *op* and production p that it can be applied to, TR required to kill mutant from p

8.2.3 Weak and Strong

Strong Mutation : fault must be reachable, infect state, and propogate to output

Weak Mutation : fault which kills the mutant needs to only be reachable and infect state

Strong Mutation Coverage : for each mutant, TR has a test that strongly kills it

Weak Mutation Coverage : for each mutant, TR has a test that weakly kills it

8.2.4 Mutation Operators

- Absolute value insertion: $x > a \Rightarrow x > abs(a)$
- Operator replacement: $x > a \Rightarrow x < a$
- Scalar variable replacement: $x > a \Rightarrow x > b$
- Crash statement replacement: ...

8.2.5 Integration Mutation

- Change calling method's parameters
- Change method being called
- Change inputs and outputs of called method

9 Logic Coverage

9.1 Introduction

Cover the **predicates** (expression that evaluates to a boolean value) by covering the **clauses** (predicate without logical operators) they're made of. Clauses are connected by **logical operators**:

- $\neg$ negation
- $\vee$ or
- $\oplus$ exclusive or
- $\wedge$ and
- $\implies$ implication
- $\iff$ equivalence

9.2 Basic Coverages

Predicate Coverage : for each predicate p, cover $p = true$ and $p = false$ (analagous to CFG edge coverage)

Clause Coverage : for each clause c, cover $c = true$ and $c = false$

Combination Coverage : for each predicate p, cover all possible truth values for clauses in C_p

9.3 Active Clause Coverage

We want to make a **major** clause determine the predicate, by setting the other (**minor**) clauses. If $p_{c=true}$ represents p with $c = true$ and similarly $p_{c=false}$ then p_c describes the conditions necessary for c to determine p

$$p_c = p_{c=true} \oplus p_{c=false}$$

Example 9.1. Determining a predicate

$$\begin{aligned} p &= a \vee b \\ p_a &= (true \vee b) \oplus (false \vee b) \\ &= true \oplus b \\ &= \neg b \end{aligned}$$

Therefore a determines p when $b = false$

Active Clause Coverage : for every major clause $c_i = true$ and $c_i = false$, cover all assignments of minor clauses so that c_i determines p

General ACC : ACC without restrictions on minor clauses, they may be different for $c_i = true$ and $c_i = false$

Correlated ACC : ACC but minor clauses must cause $p = true$ for one value of c_i and $p = false$ for the other

Restricted ACC : ACC but minor clauses must be the same for $c_i = true$ and $c_i = false$

9.4 Feasibility

Sometimes TRs can be infeasible, this can happen for many reasons:

- predicate is unreachable
- predicate never has appropriate values to cause desired clause values *e.g. len(array)<0*
- clause never determines predicate and you're attempting ACC

Therefore to get logic coverage:

1. identify the predicates
2. figure out how to reach each predicate
3. make c determine predicate p
4. find values for program vars to meet criteria

10 Reporting Bugs

Bug reporting help fix bugs faster and improve software dev cycle. Good bugs that should be reported are:

- sufficiently general
- have severe consequences
- affect most recent version

Good reports allow developers to quickly understand and solve the bug:

- reported in the database
- simple: one bug per report
- understandable, minimal, generalizable
- reproducible
- non-judgemental
- not duplicates

11 Input Space Partitioning

11.1 Introduction

We don't have time to test all inputs, so we want to test one input from each representative partition.

Characteristics : how values are distinguished

Partition : a way of splitting values into blocks

Block : a set of values similar by characteristics

Partition should be:

- **Complete**: covers the entire domain
- **Disjoint**: no overlap

11.2 Input Domain Modelling

1. find units/functions to test
2. identify parameters of each unit
3. come up with input space model

Two approaches:

Interface-based: just using input space (without considering how it is used)

+ easy to identify characters
+ easy to translate test cases
- less effective because it doesn't use domain knowledge *e.g. variable relationships*

Functionality-based: using a functional view of the program (how the program works)

+ may have better test cases because of domain knowledge
+ can create models just from specifications (without seeing actual code)
- may be hard to identify characterstics, values
- harder to generate test from this IDM

11.3 Identifying Characteristics

Some possible characteristics:

- preconditions and postconditions
- relationships between variables
- extract characteristics from specifications

11.4 Choosing Blocks and Values

- both valid and invalid values
- boundary and non-boundary values
- special values

An **alternate approach**, instead of multiple blocks, is to choose True/False characteristics. This dichotomy guarantees disjointness and completeness.
We could also use **multiple IDMs**, one for valid values and one for invalid values.

11.5 Combining Characteristics

We want to test multiple inputs/characteristics, each with their own partitions

All Combinations Coverage : all combinations of blocks from all characteristics (exhuastive, but too big!)

Each Clause Coverage : at least one value from each block for each characteristic

Pair-Wise Coverage : combine a value for each block for each characteristic with some value from every other block

T-Wise Coverage : same as pair-wise for T values from other blocks for other characteristics

Drop infeasible TRs

11.6 Base Choice

We have been treating all blocks as equal, but we can designate one block as the most important called the **base choice** and by choosing the base choice for each characteristic we have a **base test**

Base Choice Coverage : vary the base test one characteristic at a time with all other blocks

Multiple Base Choice Coverage : use more than one base choice block for each characteristic and then vary

If TRs are infeasible, change base case

12 Regression Testing

12.1 Overview

Regression Testing: re-testing modified programs to uncover regression, usually with integration test

- automated: with git hooks, push process ...

- appropriately sized: too small and it is useless, too large and it takes too long
- up-to-date: tests are valid for the version of the program being tested

Good TODOs:

- remove redunant/old test cases over time
- evaluate test cases manually or with coverage criteria
- test case prioritization is good but implementation is unclear

12.2 Output Validation

Regression tests have a low yield for bug-finding, so automation is very important.

- input: file input is easiest, but we may need to create special mocks
- output: verifying output should account for resolution, whitespace ...
- UI: capture and replay events, use Selenium for web dev

To verify our output is correct:

- manual effort
- compute multiple ways *e.g. sort differently*
- checksums/redundant data

12.3 Best Practises

Unit Tests : each class has associated unit tests

Code Reviews : code must be approved by "owners" of that branch/block

Continuous Builds : continuously checkout and build latest code to enforce good practises

One-button Deploy : if all test have passed, deploying should be easy

Back Button : systems should have easy rollback in case of errors

13 Badly Designed Tests

Smell: a symptom of a problem implying something is wrong

13.1 Project Smells

Highest level smells, usually detected by project managers

- production bugs: too many production bugs implies bad testing, development process, resources ...
- continuous integration failures: could be buggy, expensive, or insufficient tests

13.2 Behaviour Smells

Typically manifest as compile errors or test failures

- fragile tests: tests too sensitive to interface, data, context, or behaviour
- assertion roulette: integration failures with bad error messages
- erratic/flakey tests: could be caused by external dependencies
- frequent debugging: insufficient or not fine enough tests
- slow tests
- tests requiring manual intervention

13.3 Code Smells

Affect maintenance costs and are also early warning signs of behaviour smells

- obscure tests: caused by poorly-named or implemented tests
- conditional test logic
- hard-coded test data
- hard-to-test code
- test code duplication
- test logic in production