CSCB07 - Software Design

Software Testing

Summer 2021

What is Software Testing?

- Running a program in order to find faults
  - ➢ Examining the code without execution is not testing
- The main practical approach to validate/verify software
  - ➢ Formal methods that aim at proving the correctness of a program are not scalable
- “Program testing can be used to show the presence of bugs, but never to show their absence!” — Edsger W. Dijkstra

Testing Levels

- Acceptance testing
  - Test whether the software is acceptable to the user
- System testing
  - Test the overall functionality of the system
- Integration testing
  - Test how modules interact with each other
- Module testing
  - A module is a collection of related units the are assembled in a file, package, or class
  - Test modules in isolation including how the components interact with each other
  - Responsibility of the programmer
- Unit testing
  - Test units (methods) individually
  - Responsibility of the programmer

Black-Box and White-Box Testing

- Black-Box Testing
  - Test are derived from external descriptions of the software
- White-Box Testing
  - Test are derived from the source code internals of the software
  - More expensive to apply

Why is Software Testing Hard?

- Exhaustive testing is infeasible
  - E.g. Exhaustively testing a method with two integer parameters would require ~10^{19} tests
- Random/statistical testing is not effective

Why Do We Test Software?

- Software is everywhere
  - Communication, transportation, healthcare, finance, education, etc.
- Software failures could have severe consequences
  - A 2002 NIST report estimated that defective software costs the U.S. economy $59.5 billion per year and that improvements in testing could reduce this cost by about a third
  - In certain areas such as healthcare and transportation, software failures could cost lives

Infamous Software Failures

- Northeast blackout of 2003
  - Caused by a failure of the alarm system
  - Affected 40 million people in USA and 10 million people in Canada
  - Contributed to at least 11 deaths
  - Cost around $6 billion
- Ariane 5 explosion (1996)
  - Unhandled floating point conversion exception
  - Estimated loss: $370 million
- NASA's Mars lander (1999)
  - Crashed due to an integration fault
  - Estimated loss: $165 million

Infamous Software Failures

- Boeing 737 Max
  - Crashed due to overly aggressive software flight overrides
- Boeing A220
  - Engines failed after software update allowed excessive vibrations
- Toyota brakes failure
  - Dozens dead
  - Thousands of crashes
- Therac-25 radiation therapy machine
  - Three patients were killed

Fault/Error/Failure

- ***Software Fault***: A static defect in the software
- ***Software Error***: An incorrect internal state that is the manifestation of some fault
- ***Software Failure****:* External, incorrect behavior with respect to the requirements or another description of the expected behaviour
- The term ***bug*** is often used informally to refer to all three of fault, error, and failure
  - The first computer bug was an actual bug!

Fault/Error/Failure (Example)

```
public static int numZero (int [ ] arr){
        int count = 0;
        for (int i = 1; i < arr.length; i++){
                if (arr [ i ] == 0){
                        count++;
                }
        }
        return count;
}
```

Fault: Should start searching at 0, not 1

Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Error: i is 1, not 0, in the first iteration
Failure: none

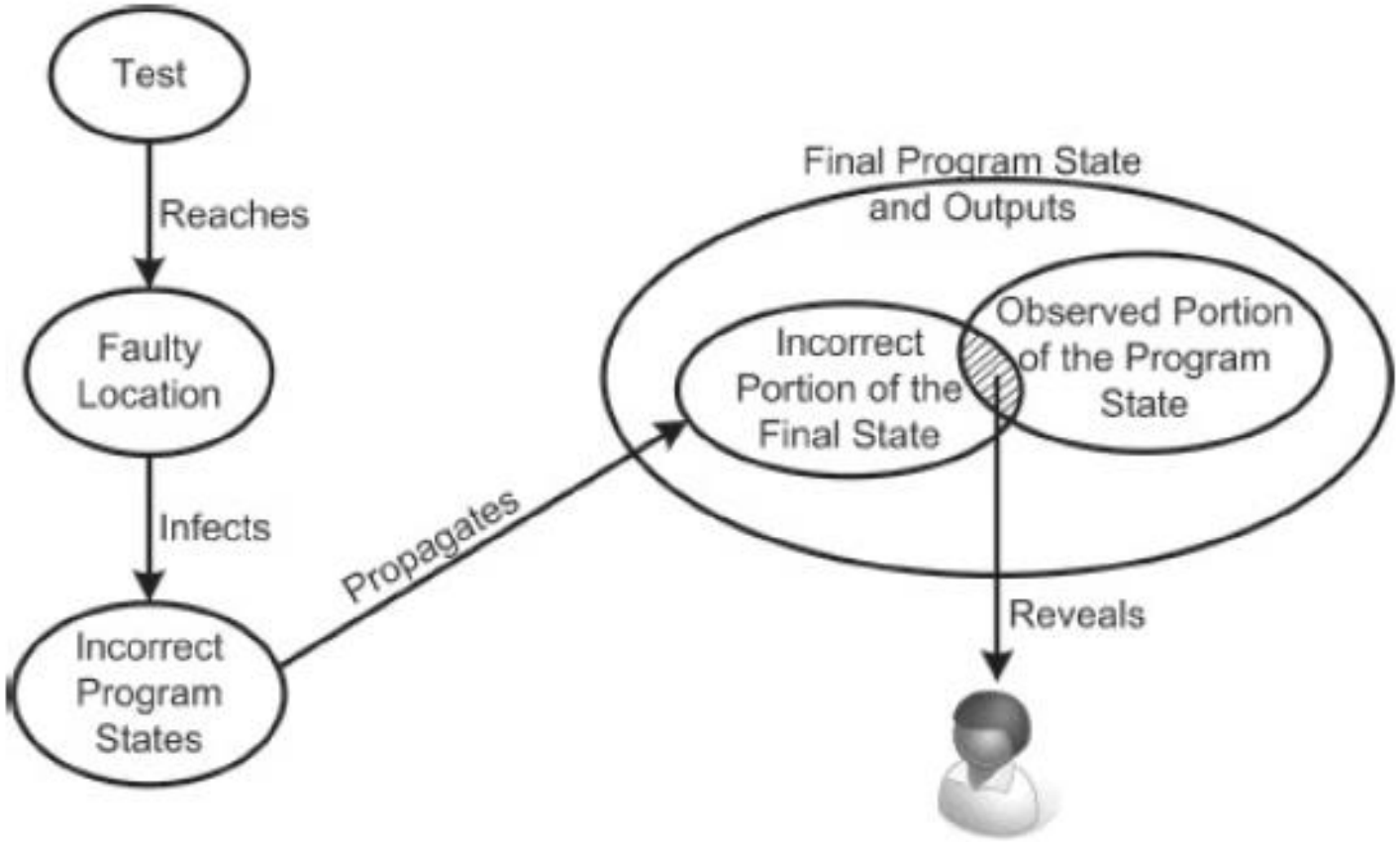
Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Error: i is 1, not 0, in the first iteration
Error propagates to the variable count
Failure: count is 0 at the return statement

The RIPR model

- Four conditions are needed for a failure to be observed
  1. ***Reachability***: a test must reach the location in the program that contains the fault
  2. ***Infection***: After the faulty location is executed, the state of the program must be incorrect
  3. ***Propagation***: The infected state must propagate through the rest of the execution and cause some output or final state of the program to be incorrect
  4. ***Revealability***: The tester must observe part of the incorrect portion of the final program state

The RIPR model



Ammann and Offutt, Introduction to Software Testing, Second Edition, 2017

Criteria-based Test Design

- ***Coverage Criterion****:* A rule or collection of rules that impose test requirements on a test set
  - E.g. For each statement in the code, there should be at least one test case that covers it
- Coverage criteria give us structured, practical ways to search the input space. Satisfying a coverage criterion gives a tester some amount of confidence in two crucial goals
  1. We have looked in many corners of the input space, and
  2. Our tests have a fairly low amount of overlap
- Criteria subsumption
  - C_1 subsumes C_2 if and only if every test set that satisfies C_1 satisfies C_2

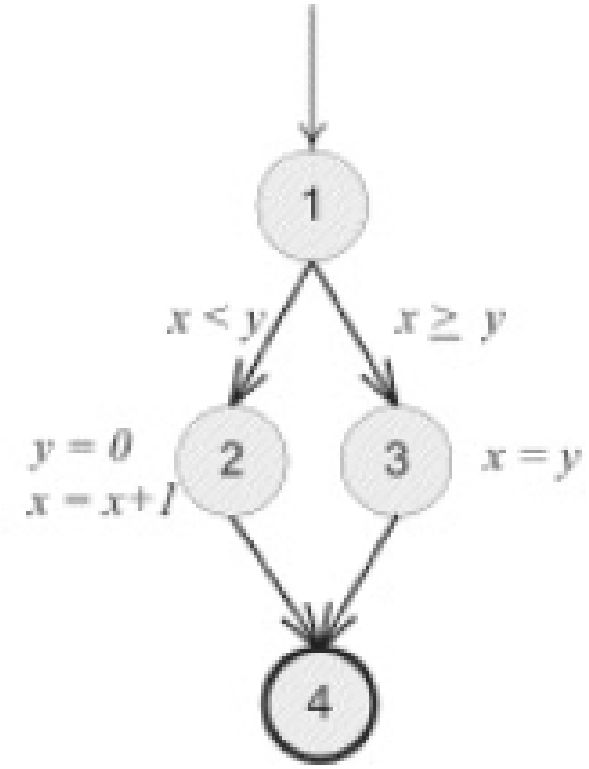
Criteria-based Test Design
Graph Coverage

- The software is modeled as a graph where nodes and edges could represent:
  - Methods and calls
  - Statements and branches
  - Etc.
- Coverage criteria are defined based on the graph. For example:
  - Cover every node
  - Cover every edge
  - Cover every path
  - Etc.

Criteria-based Test Design
Graph Coverage

- Example (Control Flow Graph)

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



Criteria-based Test Design
Logic Coverage

- Involves the boolean expressions of the code
- Coverage criteria include:
  - Predicate coverage
  - Clause coverage
  - Combinational coverage
  - Etc.

Criteria-based Test Design
Logic Coverage

- Example

```
if(((a>b) || c) && (x<y))
        …
else
        …
```

- Predicate coverage
  - The test set should make each predicate evaluate to true and false
  - E.g. **((a>b) || c) && (x<y) = {True, False}**
- Clause coverage
  - The test set should make each clause evaluate to true and false
  - E.g. **(a>b) = {True, False}, c = {True, False}, (x<y) = {True, False}**

Criteria-based Test Design
Logic Coverage (Active clause coverage)

- Clause coverage has a weakness
  - The values do not always make a difference
- Active clause coverage
  - A clause c_i in predicate p, called the major clause, determines p if and only if the values of the remaining minor clauses c_j are such that changing c_i changes the value of p
  - Two requirements for each c_i: c_i evaluates to true and c_i evaluates to false
  - This is a form of MCDC, which is required by the FAA for safety critical software

Criteria-based Test Design
Logic Coverage (Inactive clause coverage)

- Ensures that "major" clauses do not affect the predicates
- Four requirements for each c_i
  1. c_i evaluates to true with p true
  2. c_i evaluates to false with p true
  3. c_i evaluates to true with p false
  4. c_i evaluates to false with p false
- Example
  - Testing the control software for a shutdown system in a reactor where the specification states that the status of a particular valve (**open** vs. **closed**) is relevant to the reset operation in **Normal** mode, but not in **Override** mode

Test Oracles

- A *test oracle* is an encoding of the expected results of a given test
  - E.g. JUnit assertion
- Must strike a balance between checking too much (unnecessary cost) and checking too little (perhaps not revealing failures)
- What should be checked?
  - The output state is everything that is produced by the software under test, including outputs to the screen, file, databases, messages, and signals
  - Each test should have a goal and testers should check the output(s) that are mainly related to that goal
    - At the unit testing level, checking the return values of the methods and returned parameter values are almost always enough
    - At the system level, it is usually sufficient to check the directly visible output such as to the screen

Test Oracles

How to determine what the correct results are?

- Specification-Based direct verification of outputs
  - E.g. “a ***sort*** program should produce a permutation of its input in increasing order”
  - Specifications are hard to write
- Redundant computations
  - Refer to another trustworthy implementation of the program
  - Usually used for regression testing
- Consistency checks
  - Check whether certain properties hold (e.g. a value representing probability should neither be negative nor larger than one)